НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено
Завідувач кафедри
_____ О.В. Коваль
(підпис) (ініціали, прізвище)
“___”_____2020р.

ДИПЛОМНА РОБОТА

на здобуття ступеня бакалавра

з напряму підготовки 121 Інженерія програмного забезпечення

на тему: “ШВИДКЕ ПЕРЕТВОРЕННЯ ФУР’Є В ЗАДАЧАХ ОБРОБКИ ТЕКСТІВ”

Виконав: студент 4-го курсу, групи ТВ-з61

Джулай Володимир Васильович _____
(підпис)

Керівник кандидат наук, доцент Стативка Ю.І. _____
(підпис)

Консультант _____ _____ _____
(назва розділу) (вчені ступінь та звання, прізвище, ініціали) (підпис)

Рецензент_____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів
без відповідних посилань.
Студент _____
(підпис)

Київ – 2020 року

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший рівень

Напрям підготовки: 121 Інженерія програмного забезпечення

Спеціалізація: Програмне забезпечення розподілених систем

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ О.В. Коваль
(підпис)
”___”_____2020р.

ЗАВДАННЯ
на дипломну роботу студенту

Джулаю Володимиру Васильовичу

1. Тема роботи: “ШВИДКЕ ПЕРЕТВОРЕННЯ ФУР’Є В ЗАДАЧАХ ОБРОБКИ ТЕКСТІВ”

керівник роботи: Стативка Юрій Іванович, кандидат наук, доцент

затверджена наказом вищого навчального закладу від ”__”___ 202__р. № __

2. Строк подання студентом роботи: 14.06.2020
3. Вихідні дані до роботи: ОС Ubuntu, мова С++

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): Дослідити наявні алгоритми пошуку в тексті за зразком; Виявити недоліки та незручності їх застосування; Продумати логіку використання швидкого перетворення Фур’є для обчислення конволюції многочленів та подальшого виявлення точок повних збігів між рядком у тексті та зразком; Розробити програмну реалізацію цього алгоритму засобами мови Сі++; Зробити програму зручною для подальшого тестування.

5. Перелік ілюстративного матеріалу

_____
_____
_____
_____
_____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання ”11” жовтня 2020 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Затвердження теми роботи		
2.	Вивчення та аналіз задачі		
3.	Розробка архітектури та загальної структури системи		
4.	Розробка структур окремих підсистем		
5.	Програмна реалізація системи		
6.	Оформлення пояснювальної записки		
7.	Захист програмного продукту		
8.	Передзахист		
9.	Захист		

Студент _____ Джулай В.В.
(підпис)

Керівник роботи _____ Стативка Ю.І.
(підпис)

АНОТАЦІЯ

Дана пояснювальна записка містить 70 сторінок без врахування додатків, в неї включені 2 діаграми, 1 таблиця, 1 основоположне твердження з доведенням, на якому базується робота алгоритму, та 27 посилань. Мета роботи полягала в реалізації пошуку в тексті, використовуючи можливості обчислень швидкого перетворення Фур'є. В роботі розглянутий метод швидкого множення многочленів за допомогою швидкого перетворення Фур'є з подальшим використанням знайдених значень для пошуку збігів між частиною тексту та шуканим зразком. Вибір даного методу полягав у його стабільній швидкодії на великих масивах тексту та зразка. В ході роботи реалізовано програму, яка виконує пошук зразка в тексті за точним співпадінням, з використанням перетворення вхідних даних методом швидкого перетворення Фур'є.

Ключові слова: перетворення Фур'є; пошук зразка в тексті; обробка текстів; математичний алгоритм.

ABSTRACT

This dissertation examines the pattern search algorithm based on Fast Fourier transform method. Using the mathematical processing of text characters, this approach could become a powerful tool for pattern searching in the case of large observed text and search pattern both. Main method of the work is to get text-and-pattern convolution results with Fast Fourier transform use and to find matching points with pivot match value. There was developed the program that uses Fast Fourier transform and finds all the exact matches between some part of text and the pattern that this program searches for.

Keywords: Fast Fourier transform; Mathematical approach algorithm; Pattern search algo; Text processing.

ЗМІСТ

Вступ

Функція пошуку в тексті за зразком є однією з найосновоположніших для будь-якого текстового редактора, так як дана функція автоматизовує одну з найперших людських потреб при роботі з текстами — швидке знаходження певного слова або словосполучення у великому тексті. Так як в сучасному світі тексти досягають величезних розмірів, пошуки відповідних словосполучень мають ставати дедалі ефективнішими та швидшими. Більше того, в сучасній науці, зокрема в мікробіології, великим може бути не лише вхідний текст, а й зразок для пошуку, наприклад - пошук відповідних для порівняння ділянок в буквенній інтерпретації складових амінокислот геному живих істот. А порівняння ділянок геному з цілим масивом генетичних кодів певного ряду прикладів вимагає значного рівня прискорення обчислень.

Метою даної роботи є створення програми, яка повинна виконувати швидку перевірку тексту та зразка на входження, за умови, що і текст, і зразок будуть подані у вигляді текстових файлів. Ця перевірка повинна бути здійснена на основі математичного перетворення двох числових масивів, - інтерпретацій символьних масивів - в результаті якої виникає нова числова послідовність, з якої легко (або відносно легко) можна отримати інформацію стосовно входження або невходження шуканого зразка в текст, та знайти імовірні точки входження.

Актуальність розробок стабільного та швидкодійного алгоритму, базованого на використанні математичного підходу, зберігається. До того ж, ця робота могла б стати в нагоді розробникам, які могли б продовжити математичні дослідження або знаходженням ефективнішого алгоритму на основі представленого в цій роботі.

1. Пошук в тексті заданого зразка

Пошук в тексті за зразком є, напевно, одним з найвикористовуваніших функцій в програмах, якими люди користуються щодня. В разі потреби знаходження певного місця в тексті, де йде мова про певну тему, комп’ютерні користувачі звертаються до функції пошуку з використанням визначеного ними ключового слова, яке є або еквівалентом цікавлячого користувача поняття, або, як мінімум, максимально наближеним до схоплення бажаної інформації поняттям. Класичний пошук в тексті є реалізацією методу знаходження визначеного набору в точно тому ж порядку, серед масиву символів, тобто в певному наборі текстової інформації. Поглиблений пошук інформації полягає в створенні складних систем, чия робота базується на принципах машинного навчання, з допомогою чого стає реальною можливість пошуку не лише за точним співпадінням, але й по “смислу”, тобто знайдена інформація є більш релевантною для шукача, аніж багато інших матеріалів, де присутнє точне співпадіння символів.

Однак подібний алгоритм пошуку базується на співставленнях наборів символів за їх близькістю та взаємозамінністю, частотою використання, відмітками та відгуками самих користувачів стосовно корисності інформації. Але запорукою проведення поглибленого аналізу інформації є базовий метод - пошук точного співпадіння в тексті за зразком.

Складність проведення пошуку за зразком зростає в прямій пропорційності з розміром шуканого зразка та оброблюваного тексту. Якщо користувачу необхідно знайти невелике слово у малому за обсягом тексті, то такий пошук відбудеться з допомогою сучасної обчислювальної техніки майже миттєво. Однак, завдання значно ускладниться, якщо завдання, тобто вхідні дані, будуть задаватися не людиною як користувачем системи, а іншою системою, тобто надходити як вхідні дані в пошукову програму з

іншої програми або системи, складність якої та масивність даних можуть бути переважаючими можливості обчислювальної системи, яка проводить пошук.

Галузями, де активно застосовуються алгоритми пошуку саме на великих масивах інформації, є пошук у великих масивах природномовного тексту, пошук в біологічних/геномних базах даних, в системах контролю версій тощо.

Для текстів природних мов ускладненням може бути точний пошук великого зразка, наприклад, кількох абзаців тексту серед великої кількості текстів, які подаються як один великий текстовий масив. Оскільки пошук проводиться для великого зразка, кількість входжень навіть серед значного текстового масиву буде мінімальною. Однак сам пошук точного співпадіння кількаабзацевого зразка серед багатьох текстових творів може бути дуже масивним завданням. Використання в молекулярній біології полягає в тому, що біологічні молекули можуть бути представлені як послідовності нуклеотидів або амінокислот. Кількість інформації в галузях текстів людських мов та наборів молекулярних та біологічних даних має тенденцію до збільшення в два рази кожні 18 місяців **[2]**. Тому є вкрай необхідним розробка дедалі ефективніших алгоритмів пошуку. Складною задачею, що мусить використовувати швидкий пошук за зразком, є також пошук відповідної версії в системі контролю версій. Контролем версій називається система, що записує всі зміни, внесені до файлу або набору файлів, для можливості відтворення та повернення файлів до попередніх варіантів у разі виникнення потреби **[3]**. Такі системи використовуються розробниками ПЗ, дизайнерами та іншими користувачами професійних програм для збереження проміжних результатів, можливості повернутися до попередніх розробок, та створення проектних розгалужень, у випадку появи кількох рівноцінних підпроектів в рамках однієї системи. Кожна зміна, - коміт - вноситься до

системи контролю версій під унікальним хешованим набором значень. Власне, пошук певної версії продукту серед всього масиву раніше створених файлів теж зручно проводити через хеш-ідентифікатор. Зазвичай, такі ідентифікатори є досить великими в серйозних професійних системах контролю версій, тому ефективність пошуку за часовим параметром є вкрай важливою і для такого роду систем.

Алгоритми пошуку в тексті за зразком також слугують як базові парадигми в інших галузях розробки програмного забезпечення, що мають суміжні проблеми та задачі, пов’язані з точним співпадінням певного набору значень, виставлених у певному порядку. В роботі “Співпадіння квантового зразка” (Quantum search maycching) щодо проблематики квантових обчислень, презентованій в 2005 році, автори П. Матеусом та Й. Омаром, зазначають, що пошук зразка в базах даних має найпоширеніше та найвживаніше застосування в комп’ютерній науці, так як кожен користувач шукає слово в тексті, або шукає відповідний текст за ключовими словами в пошукових системах на кшталт Google **[2]**. Входження в еру квантових обчислень потребує також продовження роботи над пошуками оптимальніших алгоритмів для складних систем, які, звісно, зовсім не обов’язково будуть такими ж ефективними для простіших пошуків.

Метою роботи є пошук застосування числового алгоритму для пошуку точних співпадінь в тексті. Використання числового алгоритму є математичною обробкою представленої у числовому вигляді текствової інформації, яка на виході дає комплект значень, з яких за відносно дешеву в обчислювальному плані операцію можна отримати інформацію про входження або невходження зразка в оброблюваний текст, та вказувати на номери місць у тексті, якщо пошук був успішним. Однією з підзадач є пошук в тексті зразка з неважливими символами, тобто такими, які співпадають з будь-якими іншими символами в тексті. Цей підхід є корисним у випадку

пошуку певної частини зразка, у поєднанні з іншою частиною зразка, без різниці, які символи можуть зустрічатися між цими двома частинами. За основну математичну модель, за допомогою якої ми маємо намір добитися знаходження алгоритму ефкетивного пошуку, є модель швидкого перетворення Фур'є, тобто прискореного пошуку значень ряду Фур'є для многочлена. Цей підхід вже давно розглядався в якості можливого варіанту проведення пошуку в тексті. Зокрема, цей підхід був побіжно описаний в книзі Дена Гасфілда "Ряди, дерева та послідовності в алгоритмах. Інформатика та молекулярна біологія" **[1]**. Однак досі він не став широковикористовуваним. Тому, окрім розробки самої математичної моделі та програмної імплементації, потрібно провести додаткові тестування даного підходу, порівнявши результати з вже існуючими та добрезарекомендеваними алгоритмами, що використовуються для вирішення тих же задач.

Завданням даної роботи є розробка програми, яка ефективно використовує математичну модель швидкого перетворення Фур'є, втілюючи її в програмному алгоритмі, на основі якого відбувається ефективний пошук точок збіжності між шуканим зразком та оброблюваним текстом. Для цього, в першу чергу, необхідно провести аналіз відомих програмних рішень в цій галузі, таких як алгоритм Кнута-Морріса-Пратта та алгоритм Рабіна-Карпа. Потім потрібно віднайти оригінальне рішення застосовності результатів швидкого перетворення Фур'є для задачі пошуку, довести таку можливість та показати адекватність результатів. Останнім етапом є власне створення програми, реалізуючої даний алгоритм, причому потрібно провести максимальне спрощення, оптимізацію підходу для знаходження максимально вигідного ходу програми. Ця програма може бути базовою для подальшої імплементації алгоритму швидкого перетворення Фур'є для пошуку в тексті за зразком та для інших задач розбору тексту. Алгоритм може бути не реалізованим на тому рівні, щоб він міг сходу стати конкурентним порівняно

з простішими в математичних розрахунках та ефективнішими алгоритмами пошуку, що реалізовані в текстових редакторах всього лише як один з елементів функціоналу. Тим не менше, обробка великих масивів текстової інформації потребує нових підходів, так як простіші механізми не можуть дати комплексних рішень прориву в швидкодії саме на великих масивах інформації. Можливості цього методу є недостатньо вивчені, при подальшому аналізі та розробці можливо вийти на рівень кращого алгоритму для роботи з великими текстами. Програма також має бути побудована таким чином, що дозволило б проводити порівняльні тести з іншими алгоритмами пошуку. Основними користувачами програми на перших етапах вбачаються інші програмісти, які можуть провести додатковий аналіз та запропонувати свої варіанти покращення цього підходу для подальшої роботи з ним спеціалістам по обробці значних масивів інформації.

Оскільки основна суть роботи полягає в знаходженні алгоритму, то реалізація програми задумана як консольний застосунок, без додаткового графічного інтерфейсу.

Вхідні дані тексту та зразка надходитимуть в програму з файлів, які будуть розташовані в основній директорії файлів користувача **/home/ %username%/** на машині з установленою операційною системою Ubuntu. Програма очікує на файли з чіткими визначеними наперед назвами. Тобто користувач програми мусить спершу розподілити текст для обробки в файл під назвою **fourier_search_text.txt**, а шуканий зразок помістити у файл **fourier_search_pattern.txt**. У випадку відсутності файлів, або надання їм неправильних імен програма повинна видати в терміналі свого запуску повідомлення про помилку та аварійно закритися. Розміщення тексту в файл є нормальною практикою для пошукових програм та текстових редакторів. В той же час, шуканий зразок, як правило, задається динамічно, на ходу, користувачем, відповідно до його запитів. Але динамічне задавання зразка є

обмеженим у розмірах, тому ця реалізація підходить лише для невеликого розміру рядків, задаваних людиною. Реалізація пошуку через перетворення Фур’є має на меті, натомість, роботу в тому числі з великими за розмірами зразками, для яких розміщення у файл, а не пряме введення користувачами, є більш адекватним підходом.

В результаті роботи програми створюється масив індексів зустрічі шуканого зразка в тексті. Цей масив для наглядності просто відображається у вікні терміналу, де запущена програма. При бажанні, його можна перенаправити в іншу програму, або інший програмний модуль, що дозволить подальшу роботу чи аналіз із знайденим набором співпадінь.

2. Методи пошуку в тексті за зразком

2.1 Проблематика пошуку зразка в тексті та наївний підхід до вирішення задачі

Першим пунктом роботи є ознайомлення з існуючими робочими алгоритмами пошуку в тексті за зразком, та проведення підготовчого аналізу, які передумови та перетворення вхідної інформації необхідні для ефективного пошуку зразка.

Говорячи про те, як комп’ютер повинен розпізнавати входження одного рядка в інший, варто вказати, як саме відбувається кодування літер алфавітів в інформатиці. Розвиток комп’ютеризації в країнах Заходу та Америки привів до домінування латинського алфавіту, причому саме в англійському варіанті, на момент виникнення мов програмування. Для кодування інструкцій та й, власне, для оброблюваних та збережених даних використовувався саме латинський алфавіт в 32 літери. Для буквенних записів була розроблена спеціальна символьна таблиця чисел, - ASCII таблиця - де різні числа кодують різні букви. Всі літери, в регістрах великих та малих своїх відображень, розділові знаки, цифри, спеціальні символи, а також так звані керуючі послідовності, які розпізнаються процесором як інструкція для певної дії, входять в ASCII таблицю під різними номерами, та займають 1 байт інформації.

Безумовно, що значна (якщо не всі) кількість алфавітів, яка придумана людьми, зараз має підтримку графічних знаків. Закодувати всі знаки алфавітів з допомогою лише 1-го байту, неможливо, так як їх кількість значно перевищує 256, як число допустимих значень всередині 1-го байту. Тому були придумані інші таблиці для кодування літер різних алфавітів. Користуючись

ними, варто пам’ятати, що навіть літери латинського алфавіту, що вже був оброблений та виражений з допомогою ASCII таблиці, може мати іншу кодифікацію в іншій системі. Кажучи загально, поняття “простий текст” відсутнє в інформатиці. Будь-який набір символів є всього лише їх відображенням в результаті перетворення певного числа відповідно до правил тієї системи, за якою певна буквенна інформація була збережена у файл або виведена на екран. Відповідно, у звичайному текстовому файлі, створеному у програмі типу Блокнот, символи будуть закодовані у вигляді чисел-відповідників з ASCII таблиці.

Таким чином, при порівнянні двох літер на їх еквівалентність, фактично порівнюються два числа з таблиці системи, яка закодувала даний текст. Відповідно до цього порівнянням на повну еквівалентність двох однакових рядків букв, є порівняння кодуючих чисел, які стоять на однакових місцях (мають однакові індекси) у рядках, по одному. Якщо числа рівні, тобто букви однакові, то пошук продовжується. Якщо ж числа не рівні, то пошук вважається невдалим та припиняється, зафіксована невідповідність рядків один одному. Порівняння чисел здійснюється за умови однаковості чисел в різних наборах, що показують рядки, до тих пір, поки всі символи з обох рядків не буде розглянуто. Тільки проходження по всьому рядку гарантує точне розпізнавання еквівалентності рядків.

Однак частіше за все розміри рядка та тексту не збігаються, текст як правило набагато більший за розмірами за зразок. Це, власне, і є причиною використання програм, що містять такі реалізовані функції, як пошук за зразком. У випадку, коли розміри тексту переважають шуканий рядок, програма повинна відрахувати частину тексту такого самого розміру, як і шуканий рядок, та перевірити їх на збіжність. У випадку співпадіння рядка зразка та частини тексту наївний підхід до реалізації цього завдання повинен зафіксувати індекс початку збігу, що є порядковим номером серед літер

тексту першої літери співпадіння. І у випадку співпадіння, і в протилежному випадку, після співставлення частини тексту й рядка, рядок повинен бути співставлений з частиною тексту, що починається вже з другої літери тексту, а закінчується на літеру, чий індекс є на одиницю більшим від індекса останньої літери попереднього порівняння. Незважаючи на те, що більшість літер в розглядуваній частині тексту є однаковими, тим не менше їх порядок змінився, а тому кожне політерне порівняння частини тексту зі зразком є унікальним, та повинно проводитися кожного разу повністю. Наприклад, нехай у нас є масив букв txt[] = "AABC", що ми приймемо як текст; та нехай потрібно знайти в цьому масиві текстовий зразок pat[] = "ABC". В цьому випадку, стартуючи з 0-го індексу умовного текстового масиву, частина тексту, чий розмір, - 3 - буде рівним розміру шуканого зразка, матиме вигляд "AAB". Співставляючи з шуканим зразком, побачимо, що перші літери, - “A” - в обох рядках співпадають. Але вже наступне порівняння показує неоднаковість других літер "A**B**" та "A**A**". Тому проводити подальший аналіз немає сенсу — рядки не співпадають. Натомість, якщо розглянемо частину тексту "AABC", що починається з індексу 1, отримаємо "ABC". При побуквенному проходженні легко виявляється повна збіжність з рядком зразка. Таким чином, виявлено, що за індексом 1 в тексті ми зустрічаємо шуканий зразок. Оскільки стаючи на індекс 2, кількість літер в тексті стає меншою за розмір зразка, проводити подальший аналіз немає сенсу, так як повного входження зразка в текст вже не зустрінеться **[4]**.

2.2 Нечислові алгоритми

З метою вдосконалення пошуку, для скорочення кількості операцій пошуку, проводилася розробка інших методик пошуку, базованих на покроковому порівнянню символів. Дані підходи вважаються нечисловими,

так як при порівнянні не відбуваються математичні перетворення числових еквівалентів символів. Символи просто порівнюються між собою, при цьому, в загальному, навіть не має значення їх конкретне числове представлення для тієї чи іншої таблиці символів. Співставлення символів відбувається у числовій формі, але неявно, на основі функцій порівняння, закладених в стандартні бібліотеки практично кожної широковикористовуваної мови програмування. Основним прагненням покращення алгоритмів для нечислових методів пошуку є зменшення циклів порівняння рядка та тексту.

Один з найкращих нечислових методів, - алгоритм пошуку зразка Кнута-Морріса-Пратта - є значним кроком вперед порівняно з наївною реалізацією пошуку, розглянутою вище.

Говорячи дуже коротко, його суть полягає у скороченні операцій порівняння у випадку недиференційованого зразка. Як було показано на попередньому прикладі з наївного підходу, кількість операцій над двома текстовими масивами значно зростає у випадку монотонного, незмінюваного зразка, перша частина якого складається з однієї й тієї ж літери або символу. Таким чином, з кожним порівнянням та подальшим зсувом першого індекса пошуку, за умови подібної монотонності в тексті, потрібно буде проходити щоразу заново всю монотонну частину зразка, роблячи зайві порівняння **[5]**.

Тому першочерговим завданням алгоритму Кнута-Морріса-Пратта є знаходження найдовшого префікса в зразку. Створюється масив чисел довжиною в розмір зразка, де кожне число на відповідному індексу буде характеризувати, скільки одних і тих самих символів підряд зустрілися до даного індексу. Таким чином, якщо в зразку присутня послідовність одного символу, то на відповідних місцях в масиві чисел буде проходити інкрементація значень. Якщо зустрічається інший символ, то розмір префікса стає рівним нулю. Якщо знову зустрічається послідовність певного символу, інкрементовуваний рахунок знову буде відкрито. Після отримання масиву

однакових префіксів, він застосовується у випадку несходження порівнюваного символу з тексту й зразка. Суть використання полягає у переносі індекса проходження по зразку вперед на ту кількість значень, яка зазначена на відповідному індексі у масиві однакових префіксів. Таким чином, при послідовному порівнянні рядків перше неспівпадіння віднесе програму до масиву співпадаючих префіксів, і якщо виявиться, що певна кількість вже співпалих у зразкові та тексті літер є однією й тією ж літерою, то при наступному порівнянні ця частина зразка вважається такою, що співпадає, і пропускається. Виходить, що її порівнювати заново непотрібно, і порівняння починається з наступного неоднакогового з попередніми символу в масиві зразка, якщо такий, звісно, є. Якщо на місці неспівпадіння символів індекс префіксів рівний нулю, то проводиться пошук, починаючи з наступного символу тексту, та 0-го за індексом символу зразка, тобто порівняльний прогін здійснюється з самого початку. В іншому ж, при повному співпадінні частини тексту та зразка, йде звичайне порівняння числових значень символів та інкрементація індексів зразка та тексту, допоки не буде вичерпано розмір зразка. Таке проходження дасть розуміння, що програма знайшла повний збіг рядків за індексом старту даного циклу в тексті. Наприклад, якщо дано текст txt[] = "AAAAAAAB" та зразок pat[] = "AAAB", матимемо вдале порівняння перших трьох символів тексту та зразка. Наступне порівняння txt[3] з pat[3] дасть негативний результат, але буде зафіксовано значення префікса сходження в 3 символи. Оскільки перші 3 символи сходяться, то поява непідходящого під префікс символу "B", символу, на якому, власне, спільний префікс і обривається, тому числове значення додаткового масиву довжини префікса рівне 0 для символу pat[3] = "B", поява цього символу на місцях порівняння txt[2] та txt[1] явно не призведе до позитивного результату пошуків. Даний алгоритм перестрибує через наступні 3 символи, та починає порівняння з символу txt[4]. Для даного

співставлення всі 4 порівняння зі зразком буде успішним, тому алгоритм виведе значення 4 як номер індексу в тексті, де знайдене повен співпадіння з шуканим зразком.

Подібний метод зсуву на певну кількість неспівпадаючих символів використовує також метод пошуку Бойєра-Мура **[6]**. Його різниця полягає в тому, що старт проходження починається з кінця тексту та порівняльного зразка. Таким чином, при оминанні зайвих елементів вираховується суфіксальний з'їзд. Цей метод корисний для природномовного пошуку, так як у більшості мов суфікс відіграє роль визначення конкретного слова за класом. Таким чином, якщо шукане слово є прикметником, то порівнюване слово в тексті, не маючи суфікса або закінчення прикметника, не буде розглядатися як потенційний кандидат на збіг, навіть якщо це слово буде спільнокореневим з шуканим. Даний метод, тим не менше, має ту ж складність, що і розглянутий вище, якщо справа стосується пошуку в великих послідовностях інформації, що мають іншу внутрішню логіку, не пов'язану з лінгвістичною.

2.3 Числові алгоритми

Алгоритми, які базуються на певному перетворенні числових еквівалентів певної множини інформаційних символів, називаються числовими алгоритмами. Сама множина символів називається алфавітом. Від довжини алфавіту залежить кількість необхідних унікальних числових ідентифікаторів, які слугуватимуть розрізнювачами для конкретних символьних значень. Кожна система репрезентації символів використовує числові ключі для позначення символів існуючих алфавітів. Як уже було зазначено, базовою системою є ASCII-таблиця, що показує відповідність між

певним числом в межах 1 байту та символом латинського алфавіту. Однак 1 байт може містити лише 256 унікальних значень в межах. Деякі з цих значень є забронюваними для так званих командних послідовностей. Тобто якщо відповідна числова послідовність буде подана на вхід програми-ретранслятора вмісту файла у типові змінних **char**, вона буде перетворена не в текстовий символ, а в певну вказівку для оброблювача значень, наприклад, позначення кінця рядка або файлу, або перехід на новий рядок при відображенні вмісту файлу, що складається із змінних типу **char**. Тому існують розширені таблиці числових еквівалентів інших алфавітів, відмінних від латиниці. При розмірі одного символу в 2 байта його числове значення може досягати 65535. Такі розміри можуть ускладнити математичні операції з числами, тому однією з перших задач є приведення чисел до певного спрощеного вигляду. Для прикладу, якщо використовується алфавіт як скінченна множина з 32 значень, то ці значення буде зручно представити, наприклад, як числа від 1 до 32, а не числа від 97 до 122, як представлені малі букви латиниці відповідно до ASCII-таблиці.

Відповідно, подальші операції, проведені з двома масивами чисел, один з яких відповідатиме зразкові, а інший текстові, виконуються для того, щоб перетворити певний набір числової послідовності, та з допомогою нових числових значень віднайти ознаки співпадіння тексту та зразка, вказавши при цьому на номер місця в тексті, де знайдений збіг.

Одним з найпорстіших та найкорисніших числових алгоритмів є алгоритм Рабіна-Карпа. Він базується на обчисленні хеш-функції зразка та послідовних частин тексту довжиною в зразок **[7]**.

Хешуюча функція має наступний вигляд:

p = (d * p + pat[i]) % q;

У цій формулі **d** позначає кількість символів в алфавіті, для числового представлення символів під цим розуміється кількість значень для змінної

певного розміру, тобто для змінної розміром 1 байт це число буде 256. Дане значення використовується у формулі для збільшення розходження значень між одними й тими ж символами, якщо вони розташовані на різних місцях. **p** - це обчислене в попередньому циклі значення хеш-функції. Первинна його ініціалізація рівна нулю на першому проході циклу. **pat[i]** є числовим еквівалентом символу, що знаходиться в тексті або масиві під назвою **pat** під індексом **i**. **q** - це певне число, на яке здійснюється ділення для зменшення розмірів хеш-значення **p**, яке росте з кожним проходженням циклу. Для збереження унікальності значення хешу число **q** мусить бути простим. В даному випадку, воно становить 101.

Хеш-значення зразка обчислюється лише раз, з усіх наявних в ньому числових значень символів. Значення першого вікна тексту обчислюється повністю теж один раз. Далі, в процесі зсуву по тексту, значення 0-го індексу видаляється з хеш-значення, так як 0-им індексом стає той, що був 1-им індексом під час минулої ітерації. Натомість, до значення хешу додається останній символ, якого в попередній ітерації ще не було. Так як хеш-формула ускладнена для збільшення унікальності з допомагою значення **d**, є необхідність врахувати це при вирахуванні хеш-значення нового вікна в тексті. Тому формула переобчислення хешу для нового витка циклу має вигляд:

t = (d*(t - txt[i]*h) + txt[i+M])%q;

В цій формулі є присутнє значення **h**. Воно обчислюється на початку роботи функції, і дорівнює **d^(M - 1)**. З допомогою даного значення експоненціального рівня максимальної заглибленості в масив, присутність змінної **d** віднімається від текстового хешу **t**, після чого вже оновлене значеня підлягає проходженню тієї ж процедури домноження на **d** та додавання наступного за розгляданим вікном числа в текстовому масиві **txt[i+M]**. Як можна побачити, в результаті виконання операції віднімання результат хешу

може стати від’ємним, якщо це станеться, то до нього потрібно додати значення простого числа, що використовується як дільник по модулю у хеш-формулі.

Значення хеш-функції для частини тексту та зразка дає первинне розуміння, чи можуть вони теоретично співпадати. Співпадатимуть однозначно ті рядки, що міститимуть однакову кількість одних і тих же літер, причому їх порядок розміщення не матиме значення.

Внаслідок того, що одне число, складене з послідовності інших чисел, не може містити всю повноту інформації про розташування елементів у масиві, а лише говорить про присутність десь у масиві тих чи інших символів, це первинне порівняння у випадку співпадіння хеш-значень потрібно ще раз перевірити, взявши за основу наївний підхід, - тобто пройшовшись по масивах зразка та розглянутого вікна тексту символ за символом, та співставивши їх на однаковість. Наприклад, "ABC" та "ACB" вирази дадуть однакові хеш-значення, і лише при проходженні цих послідовностей символ за символом, виявиться, що на 1-му індексі одного виразу знаходиться символ "B", в той час як в іншому виразі на цьому індексі буде стояти "C", і функція не виведе підтвердження знаходження рівності рядка з вікном тексту.

До числових алгоритмів слід віднести також і алгоритм, заснований на імплементації методу швидкого перетворення Фур’є (ШПФ) для цих цілей. Швидке перетворення дає знаходження рядів Фур’є, тобто комплексних коренів многочлена для можливих коренів з мінус-одиниці. Якщо представити масив чисел, що відповідає текстовому рядку, як многочлен

$$\mathbf{X(z) = \sum^{n-1}_{k=0} x_k z^k = x_0 + x_1 z + \cdots + x_{n-1} z^{n-1}},$$

то ряд Фур’є для даного многочлена

$$\mathbf{y_j = \sum_{k=0}^{n-1} \omega_n^{jk} x_k = X(\omega_n^k)}$$

буде показником частотного входження його компонентів в многочлен **[8]**. Знаходження ряду Фур'є відкриває шлях до знаходження коефіцієнтів множення двох многочленів, що буде показано в подальших розділах.

Один із способів пропонує шукати кількість неспівпадінь на основі результатів конволюції многочленів. Стверджується, що для бінарних векторів (тобто якщо алфавіт можливих значень є 2, конволюція дасть точну кількість неспівпадінь для стартової позиції. Якщо ж результатом конволюції буде 0, то це означає співпадіння бінарних векторів.

Інший спосіб пропонує шукати почленну різницю між многочленами. Для кожного індексу k= 0,1,...,n−m знаходиться дистанція між літерами, тобто різниця їх числових значень. Сума цих дистанцій вказує на присутність або відсутність збігу для двох многочленів, один з яких є вікном в тексті, а інший — шуканим рядком:

$$\mathbf{d_k=\sum_{j=0}^{m-1}(p_j-t_{k+j})^2}$$

[8]. Цей вираз розкладається на

$$\mathbf{\sum_{j=0}^{m-1}(p_j-t_{k+j})^2=\sum_{j=0}^{m-1}p^2_j-2*\sum_{j=0}^{m-1}p_jt_{k+j}+\sum_{j=0}^{m-1}t^2_{k+j}}$$

Таким чином, перший вираз обчислюється лише раз, другий вираз обчислюватиметься по ходу зсуву по тексту, але простою операцією заміщення першого елемента на наступний за межами розглянутого вікна, подібну операцію було розглянуто при вивченні методу Рабіна-Карпа. Другий же вираз обчислюється як конволюція за допомогою методу ШПФ **[9]**.

Якщо в результаті загальна сума не дорівнює нулю, то це означає, що мінімум один із символів в масивах не зійшовся. Сума дистанцій дорівнює нулю у випадку повного посимвольного збігу.

Різновидом цього способу є метод Кліффорда та Кліффорда, який пропонує як розширення попереднього способу додати у первинний вираз

квадрату різниці множення на обидва елемента многочленів, чия дистанція обчислюється на даному кроці:

$$\sum_{j=0}^{m-1} p_j t_{i+j}(p_j - t_{i+j})^2 = \sum_{j=0}^{m-1}(p^3_j t_{i+j} - 2*p^2_j t^2_{i+j} + p_j t^3_{i+j})$$

Суть полягає в доповненні пошуку для “байдужих” елементів, які співпадатимуть з будь-яким іншим елементом в протилежному масиві. Ці елементи, зазвичай позначаються за допомогою “зірочки” **[*]**, для даного алгоритму в числовому еквіваленті отримують значення 0. Таким чином, якщо серед символів рядка або тексту буде число 0, то, відповідно до цього алгоритму, множення на цей елемент проявить для даного проходу дистанцію як 0, тобто визначить два елементи як однакові.

Запропонований в цій роботі алгоритм відрізняється від запорпонованих, так як використовує дещо інший підхід для остаточного визначення збіжності.

2.4 Порівняння складності популярних алгоритмів. Недоліки

Алгоритми порівнюються за складністю по кількості операцій, які мають бути виконані для повного порівняння всього тексту довжиною **n** з шуканим рядком довжиною **m**.

Для пошуку з наївним підходом спершу порівнюється перше вікно тексту з рядком посимвольно, тобто кількість порівнянь буде **m** разів. Незалежно від результату цього порівняння наступний цикл порівняння почнеться з наступного номера індексу в тексті. Тобто якщо перше вікно тексту буде від **0** до **m - 1**, то наступне вікно займатиме індекси з **1** до **m**, далі з **2** до **m + 1** і так далі, допоки не буде досягнуто останнього символу тексту. Таким чином, пошук за зразком закінчується не на останній літері

тексту з індексом **n - 1** (де **n** - розмір тексту), а на індексі **n - m - 1** (де **m** - розмір зразка). Найгіршим варіантом пошуку буде подібний до цього розклад букв txt[] = "AAAAAAAAAAAAAAAAA" та pat[] = "AAAAA". В такому випадку співпадіння будуть зафіксовані для всіх індексів індексу від 0 до **n - m - 1.** Тобто складність цього алгоритму буде **m* (n - m - 1).** При досить малому значенні **m** та великому значенні **n** кількість порівнянь наближатиметься до **n^2**. Це найбільша складність, так як кожен символ порівнюється з кожним, і дана функція демонструватиме погану виконуваність на великих текстах. Це досить висока громіздкість та кількість порівнянь, що обов'язково проявиться та виллється у повільну роботу програми на великих текстах. Тим не менше, у випадку повної незбіжності тексту й зразка за умови того, що зразок буде короткою довжини, результат буде кращий; наприклад, якщо зразок складатиметься всього з одного символу, якого не буде в усьому тексті, то з першої операції у вікні кожного циклу буде проявлене несходження із зразком. Тому загальна складність гарантовано буде **n** операцій.

Порівняння однакових зразків та вікон при роботі алгоритму Кнута-Морріса-Пратта проходитиме теж посимвольно, як і у випадку наївного алгоритму. Його вигода проявляється при виникненні повторів у текстах або в зразкові, що дозволить проскакувати певну кількість зайвих елементів, які функція визначить як завідомо неспівпадаючі. При повному входженні зразка в текст за всіма індексами, - txt[] = "AAAAAAA", pat[] = "AAA" - матимемо кращу складність у випадку повторів символів зразкові, що було описано вище. Але за умови різних неповторюваних символів, складність буде кращою не значно. Те ж саме стосується повного невходження в текст - txt[] = "AAAAAAA", pat[] = "BB". Якщо ж говорити про постійну складову складності, то для реалізації цього алгоритму необхідний масив розміром зі

зразок, та додаткові обчислення для цього масиву, що робить його трохи непідходящим для обчислення збіжності великих зразків.

Загалом порівняння методом Кнута-Морріса-Пратта проявляє найкращу функціональність на зразках з великою кількістю повторюваних символів, Його складність буде рівна

$$\mathbf{O = (m * s) * (n - d)}$$

де **s** це кількість співпадінь в тексті, а **d** вказує на кількість однакових йдучих підряд у зразку символів. У випадку різних елементів у зразку очевидно, що кількість входжень в текст не буде високою. Якщо таке входження буде зафіксовано на певному індексі **i** для різноелементного зразка, то це означатиме, що у попередніх вікнах тексту, що починаються від **i - 1** до **i - m - 1** включно, так само як і в наступних вікнах від **i + 1** до **i + m - 1** включно, значення співпадати не будуть, більше того, фактично з першого ж символу незбіжність рядків буде очевидною. Тому складність буде в цьому випадку

$$\mathbf{\Omega = m*s + (n - m*s),}$$

де **s** позначає кількість входжень.

Стосовно числового алгоритму Рабіна-Карпа можна відзначити, що його виконання демонструє дуже якісні результати на текстах і зразках, елементи яких є досить різними. Тоді хеш-значення двох масивів демонструватимуть різні результати. В кращому випадку повних неспівпадінь хеш-значень, через **O = (n - m) + m** операцій визначення та порівняння цих значень між собою функція завершить свою роботу. Гірший випадок буде у випадку максимальної кількості збігів хеш-функції. Тоді, відповідно до алгоритму, необхідно буде робити поелементне порівняння вікон та зразка. Якщо неспівпадіння буде наприкінці порівнюваних масивів, то кількість операцій наближатиметься до **(n - m) + m + (n - m)*m**. Те ж саме буде у випадку суцільного співпадіння зі зразком по всіх позиціях в піддослідному тексті,

програма, що використовує цей алгоритм, зробить перерахування хеш-функції по кожному вікну тексту, а потім ще зробить співставлення посимвольно кожного вікна із зразком. У випадку великого зразка та великої кількості співпадінь швидкодія програми явно втрачається. Хоча, для звичайного пошуку по зразку, заданого людиною, такого алгоритму буде більш ніж достатньо. Таким чином, є необхідність створення алгоритму, більш незалежного від розміру зразка та кількості можливих входжень.

Стосовно методу ШПФ, маємо відзначити, що він належить до алгоритмів divide and conquer. Тобто він виконується шляхом рекурсивного поділу числового масиву на 2 підмасиви з виконанням певних операцій допоки в підмасивах кількість елементів не стане меншою від одного. Тому множення многочленів на основі ШПФ має обчислювальну складність **O((N+M)log(N+M)) [9]**. На основі множення двох многочленів базуються кілька алгоритмів пошуку збіжності рядків. Пошук кількості неспівпадінь для двох числових векторів має складність **O(log|Σ|nlogn)**, де **Σ** - позначає кількість елементів в алфавіті даних числових послідовностей **[9]**. Другий підхід, що визначає суму дистанцій між елементами, шукаючи співпадіння

$$\sum_{j=0}^{m-1} p^2_j - 2 * \sum_{j=0}^{m-1} p_j t_{k+j} + \sum_{j=0}^{m-1} t^2_{k+j}$$

містить три вирази у загальній формулі. Перший вираз обчислюється за **O(m)** операцій, третій вираз обчислюється за **O(n)**, другий же вираз є множенням многочленів, з допомогою розв’язання конволюції методом ШПФ його складність, як стверджується, рівна **O(nlogm)**, звідки береться, що складність всього алгоритму є рівною **O(nlogm) [8]**. Таку ж складність має також і алгоритм Кліффорда та Кліффорда **[10]**. Однак, хоча кількість циклів дійсно рівна **3*O(nlogm)**, тим не менше кількість математичних операцій суттєво виростає. В кожному виразі присутнє піднесення мінімум одного елемента до степеня. Окрім цього, піднесення до 3-го степеня з подальшою конволюцією масивів за участі отриманого результату може приводити до

досить громіздких обчислень. Застосувавши алгоритмічну оптимізацію, щоб уникнути зайвих перераховувань раніше порахованих елементів, у функції з’явиться ще один цикл для **n** та **m** елементів тексту та рядка відповідно, з втіленням відразу двох операцій — пошуку квадрату та кубу кожного з значень. А також необхідно буде створити ще по 2 додаткові масиви для обох тексту та шуканого рядка, - з метою збереження квадратів і кубів елементів.

Заявлена складність **O(nlogm)** є дуже обнадійливим результатом, тим не менше, потребується висока складність оптимізації для наближення до такого результату, до того ж висока щільність математичних операцій є не завжди виправданою, так як в результаті після всіх необхідних обрахувань все одно отримується число, яке є показником збіжності всього лише для одного індексу. Для всіх індексів тексту складні операції повинні бути виконаними знову й знову. Більш корисним, як здається, є дослідження алгоритму на основі ШПФ, який, відпрацювавши один раз, може відразу вказати на індекси збіжності або на їх відсутність.

3. Метод пошуку у тексті з використанням швидкого перетворення Фур'є

3.1 Підготовка даних

Раніше вже було показано, як саме символи алфавіту виражаються через числові значення. Саме ці числа-еквіваленти зберігаються в тимчасовій пам'яті комп'ютера та постійних носіях пам'яті. Вони ж математично обробляються у випадку застосування числових алгоритмів для певних функціональних цілей, в тому числі, - для методу пошуку збігів у тексті. Числа для позначеннння латинських літер, цифр та спеціальних символів у найпростішій таблиці відповідностей, - ASCII-таблиці - починаються з "пустого" символу (прогалини) з числовим значенням 32 в десятковій системі, та завершується символом "тільда" (~), чиє значення є 126. Операції математичних обчислень, особливо множення та піднесення до степеней, легше проводити на менших числах, так як результуючі значення виростуть не настільки, як у випадку з більшими числами. Тому, при первинній підготовці тексту, доцільним для громіздкого числового способу обробки, є зменшення номінальних значень до мінімальних. Як правило, пошук є байдужим до регістру літер, тому обробка повинна вирівняти до однакових значень великі та малі алфавітні еквіваленти. Програма не повинна уловлювати різницю між "A" та "a", хоча в ASCII-таблиці це 2 різних числа, - 65 та 97. Тепер кілька слів щодо доцільності числових перетворень в бік зменшення номіналу, та що робить їх взагалі реальними. Як було відмічено в попередніх розділах, ASCII-таблиця містить додаткові текстооперуючі елементи, керуючі послідовності. Вони займають першу частину таблиці, від 0-го символу до 31-го включно. На текстовий вигляд більшість з них не

впливає, і ніяк спеціально в тексті не проявляється, тому на них можна не звертати увагу. Таким чином, їхні числові еквіваленти вивільняються для їхнього заміщення. З іншого боку, одна з керуючих послідовностей 0x0A (**‘\ n’)** позначає перенесення курсора на новий рядок вниз. Цей символ зустрічається дуже часто тому його варто зберегти, хоча б і з іншим унікальним ідентифікатором. В даному випадку, 27 латинських літер зручно представити в числах від 0 до 31, а керуючій послідовності 0x0A можна дати унікальне значення, перенісши її до наступного півбайту, шляхом додавання її звичайного ASCII значення до максимального числа в півбайті, тобто до 128. Стосовно вибору мінімального значення для першої літери алфавіту, варто відзначити, що бажано, щоб цим числом були не одиниця і не нуль. Одиниця та нуль є числами, які при піднесенні до степеня не будуть змінюватись, залишаючись відповідно 1 та 0. Це може пошкодити дію числових алгоритмів, тому бажано першу літеру ставити як 2, другу як 3 і так далі. Число 0 має здатність перетворювати в себе будь-який множник. Тобто при пошукові, що базується на визначенні співпадіння між елементами як дистанції 0, поява нуля у математичному виразі приведе до результату 0, тобто покаже співпадіння символів, без залежності, який ще символ бере участь в цій операції. В зв’язку з цим, числом 0 можна перекривати присутність так званих “байдужих” символів, чиє значення в певній позиції шуканого зразка є неважливим для пошуку. Тобто під “байдужі” символи потраплять всі інші, і зразок вважатиметься співпадаючим з тим вікном тексту, де конкретні шукані символи на визначених місцях точно співпадатимуть з тими, які є у зразку. Зазвичай, позначаються такі символи через символ “зірка” (*). Для прикладу, зразок "Cv*fy" буде вбачений і при порівнянні з рядком "Cvofy", і з рядком "Cvufy", так як літера під індексом 2 не грає ролі для пошуку саме цього патерну. Символ * відображається в ASCII-таблиці як десяткове число 42. Тому при виборі режиму пошуку із

“неважливими” символами, цей символ потрібно перекладати в число 0, залишаючи звичне значення даному символу при виконанні точного, негнучкого, пошуку. Таким чином, оскільки модель відображення символів в числа детально показана, можна перейти до подальших дій, а саме створення числових рядів із строчок тексту та застосування числового алгоритму на основі швидкого перетворення Фур’є для їх обробки.

3.2 Математичний аналіз виконання швидкого перетворення Фур’є для обробки числових рядів

Масив чисел-відповідників для подальших перетворень зручно представити у формі, яка б зберігала певну унікальність позиції кожного з чисел. Таким способом є створення многочлена, де масив символьних числових значень формує коефіцієнти для ряду невідомих в експоненційному зростанні:

$$A(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$$

де a_i є числовим представленням символу, розташованого на і-тому індексі в певному рядку тексту. Шукаючи значення **A**, кожне значення коефіцієнту мусить бути помножене на значення **x**. Це правило називається правилом Хорнера. Рахуючи повну кількість всіх перемножень, отримується обчислювальна складність O(n^2). Однак ця складність може бути покращена, якщо спробувати скоротити обчислення за допомогою вже зроблених обчислень. Для того, щоб скоротити обрахунки, потрібно підібрати змінні **x** таким чином, щоб вони, піднесені до певного степеня, давали значення 1 та -1. Це дасть змогу розбити даний многочлен на два, довжиною **n/2** кожен. Таким чином кількість множень скоротиться вдвічі. Після

знаходження окремих результатів для двох підмногочленів залишиться лише скласти з них основний многочлен та обчислити його результат. Так само, як основний многочлен можна поділити на два, кожен з двох підмногочленів першого рівня в свою чергу ділиться на ще два підмногочлени другого рівня. Процес продовжується, поки в одному підмногочлені присутні більше ніж 2 елементи. Неважко побачити, що даний хід операцій притаманний алгоритмам виду “розділяй і пануй” (divide and conquer). Підбір значень **x** в певному степені, задовольняючи значення 1 та -1 відбувається підбором рішень рівняння $x^n = 1$. Це рівняння в комплексній площині має n розв’язків, ці розв’язки називаються n-коренями з одиниці, і позначаються вони наступним способом $1, \omega, \omega^2, \omega^3, \ldots, \omega^{n-1}$, де

$$\omega = e^{-2*Pi*i / n}$$

Тобто обчислюються вони з допомогою піднесення числа Ейлера до степеня подвійного числа “пі”, помноженого на комплексний корінь з -1, поділений на загальну кількість елементів многочлена. Варто вказати на те, що існує комплексний тригонометричний варіант даного виразу:

$$e^{-2*Pi*i / n} = \cos(2PI/n) + i*\sin(2PI/n) \qquad [11]$$

Виходячи з цього комплексного значення для кореня з одиниці, кожен поділ навпіл мусить здійснюватись на множники за ознакою парності та непарності індексів, які займають доданки у базовому многочлені. Наприклад, якщо **n** парне, то вираз ω^n дорівнюватиме одиниці, натомість вираз $\omega^{n/2}$ дорівнюватиме вже мінус одиниці. Відповідно, якщо робиться заміна $x=\omega^k$, то в другій, непарній, частині многочлена цей вираз буде віддзеркалено зі знаком “мінус” $\omega^{k+(n/2)} = x * -1 = -x$. **[12]** Подібний підхід дозволяє зменшити кількість операцій при обчисленні значення многочлена, ці властивості є корисними для подальшої обробки значень многочлена,

зокрема для швидкого обчислення рядів Фур'є, яке обрано як підхід до знаходження співпадаючих точок в многочленах.

Перетворенням Фур'є (або дискретним перетворенням Фур'є, скорочено ДПФ) називають функцію, суть роботи якої полягає у знаходженні всіх **n** значень многочлена, що складається з **n** елементів, для кожного з **n** коренів з мінус одиниці для многочлена такого розміру. Це сходиться з фундаментальною алгебраїчної теоремою, що говорить нам, що будь-який многочлен рівня **n** має рівно **n** коренів, для яких $p(r_j) = 0$. Причому деякі з цих коренів можуть бути ірраціональними, деякі комплексними, а деякі корені можуть повторюватися. **[11]** Перетворення Фур'є є теж рядом числових значень, а не одним конкретним числом. Кожне значення ряду Фур'є визначається як

$$\mathbf{a_j} = \sum^{\mathbf{n-1}}_{\mathbf{k=0}} \mathbf{a_k} \boldsymbol{\omega}^{\mathbf{jk}}_{\mathbf{n}}$$

для всіх $j = 0, \ldots, n-1$.

Тобто в кожній точці многочлена значення Фур'є обчислюється з допомогою сумування добутків всіх коефіцієнтів на j-тий корінь з мінус одиниці, піднесений до степеня порядку даного для даної ітерації коефіцієнта. У матричному вигляді дане перемноження коефіцієнтів многочлена на набір мінус-одиничних коренів, з результуючим вектором у вигляді ряду Фур'є, зображається так:

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \vdots \\ \hat{a}_{n-1} \end{pmatrix}.$$

Матриця степенів мінус-одиничного коренів називається матрицею Вандермонде. **[13]** Такий підхід зображення даного обчислення буде корисний при розборі зворотнього перетворення Фур'є, який буде розглянуто пізніше, та який буде корисний у подальшій підготовці даних для знаходження збігів. По факту, ДПФ дає значення многочлену

$$p(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$$

в n-них точках ω^0_n, ω^1_n , . . . , ω_n^{n-1}; іншими словами, $a\hat{}_k = p(\omega^k_n)$ для $0 \leq k \leq n-1$ **[14]**

Швидке перетворення Фур'є (ШПФ, винайдене Гауссом, перевинайдене Кулі та Тьюкі) базується на розбитті всього числового ряду коефіцієнтів на парні та непарні, з окремою їх обробкою згідно з формулою

$$p(x) = (a_0 + a_2x^2 + \cdots) + x*(a_1 + a_3x^2 + \cdots) = p_e(x^2) + x*p_o(x^2)$$ **[15]**,

де p_e позначає парну, а p_o — непарну частину многочлена. Дана тотожність дозволяє скоротити кількість операцій вдвічі. При застосуванні цієї операції до новоутворених многочленів, вочевидь, властивість зменшення операцій в два рази зберігається, що робить можливим перехід від складності O(n^2) для дискретного перетворення Фур'є до покращеної складності $O(n*\log_2 n)$. Звісно, для правильного послідовного розбиття на 2 частини необхідно, щоб загальна кількість елементів задовольняла експоненціальну функцію 2^n. Загальне застосування цього методу є обчислення частоти входжень складових величин до певної складної функції. **[14]**

Обраховуючи перетворення Фур'є звичайним способом, можна помітити підвирази, що однаково виводяться для кожного зі значень результуючого ряду Фур'є. Більше того, половина підвиразів відрізняється лише знаком арифметичної операції, так як значення n-кореня з мінус-одиниці може набувати значення -1 для парного степеня для обчислення числа з другої, непарної, половини масива. Відтак кожне число з умовно парної для даного

рівня розбиття масиву частини, взаємодіє зі своїм відповідником по місцезнаходженню в другій, непарній, частині, за допомогою лише двох операцій, як-то додавання та віднімання. Значення цих операцій заносяться як заміщення цих коефіцієнтів. **[15]**

Даний підхід для знаходження коефіцієнтів ряду Фур'є називається "метеликом", та умовно позначається як

$$a_e + a_o ; a_e - a_o$$

Перший результат обчислення заноситься відповідно до як новий коефіцієнт для парного індексу е, другий результат стає коефіцієнтом ряду Фур'є на місці індексу о. Варто відзначити, що дані коефіцієнти є тимчасовими, вони заміщують попередні коефіцієнти многочлена та використовуються для аналогічних обчислень в наступних стадіях рекурсивного підйому аж до фінального вигляду ряду Фур'є для даного многочлена. Тепер треба торкнутися питання, в якій саме послідовності застосовуватимуться операції "метелика", тобто пари коефіцієнтів на яких первинних індексах многочлена обчисляться першими та ляжуть в основу подальших обчислень, а які пари обчисляться останніми, давши остаточний результат.

При рекурсивному застосуванні розбиття на парні та непарні складові індекси на найнижчому щаблі рекурсії розміщуються у специфічному порядку. Розглянемо масив 2^4, тобто 16, елементів. Індекси цього масиву будуть розташовані звісно в порядку від 0 до 15:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Після першого розкладу на парну та непарну частину отримаються два масиви з індексами, розташованими в наступному порядку:

0 2 4 6 8 10 12 14 || 1 3 5 7 9 11 13 15

Далі кожен з цих масивів розбивається ще на два, і порядок індексів в цих масивах буде наступний:

0 4 8 12 || 2 6 10 14 || 1 5 9 13 || 3 7 11 15

Поділ цих 4-х масивів на парні/непарні компоненти дасть 8 масивів до 2 елементи, послідовність первинних індексів в них буде такою:

0 8 || 4 12 || 2 10 || 6 14 || 1 9 || 5 13 || 3 11 || 7 15

Наступне розбиття немає сенсу робити, так як кількість елементів в масиві буде менше ніж 2, математичну операцію буде неможливо зробити. Так отримується фінальний порядок розстановки елементів для масиву з 16 елементів на найглибшому рівні рекурсивного спуску. **[16]**

Саме це розбиття і є ініціюючим порядком індексів для обчислення. Цей порядок зміни порядку індексів має легкий спосіб знаходження з допомогою бінарного представлення чисел. Для того, щоб вичислити номер індексу, який повинен знаходитися на певному місці в переставленому масиві, достатньо взяти за основу бінарне число кількістю бітів найбільшого індексу в масиві коефіцієнтів многочлена, для якого здійснюється проведення ШПФ, та віддзеркалити біти, поставити їх у зворотньому порядку. Таким чином, отримається номер індексу, значення з якого повинно знаходитися на певному індексі обчислюваного масиву. **[17]**

Наостанок, варто зазначити, що значення n-кореня з мінус-одиниці змінюватиметься в два способи. Перший спосиб полягає в тому, що при рекурсивному підйомі кількість елементів підмногочленів, які будуть обчислюватися, буде різною. А оскільки в формулу визначення даного коефіцієнту входить кількість елементів в масиві, як кількість комплексних коренів для даного многочлену, то базове значення ω змінюватиметься на кожній сходинці поділу. Другий спосіб полягає в тому, що для кожного многочлена значення комплексного кореня для кожної пари елементів підноситиметься до степеня. Так, для масиву з 8 елементів визначення цього множника буде наступним:

Таблиця 1. Зміна комплексного коефіцієнту для взаємодіючих пар у масиві з 8-ми елементів

Індекс елемента в новому масиві з парної половини масиву	Індекс елемента в новому масиві з непарної половини масиву	Значення ω для пари
0	4	ω^0
1	5	ω^1
2	6	ω^2
3	7	ω^3

Процес знаходження ряду Фур'є для многочлена є зворотнім. Тобто з допомогою ряду Фур'є можна знайти початкові коефіцієнти многочлена. Причому базова формула ШПФ для знаходження зворотнього ШПФ залишається майже такою ж самою. Так, для ряду Фур'є, що отриманий з многочлена шляхом застосування ШПФ

$$X_k = \sum_{n=0}^{N-1} x_n * e^{-i2PIkn/N}$$

зворотнє перетворення Фур'є буде виглядати як

$$X_k = 1/N * \sum_{n=0}^{N-1} x_n * e^{i2PIkn/N}$$

Практична користь від зворотнього ШПФ проявляється у виведенні способу швидкого множення многочленів. При звичайному множенні многочленів кожен коефіцієнт одного многочлену буде почергово перемножений на кожен елемент іншого многочлену. Складність такого алгоритму O(n^2), що є занадто об'ємним за кількістю операцій при великих розмірах масивів. Натомість з допомогою ШПФ можна скоротити складність до $O(n*\log_2 n)$.

Відповідно до теореми конволюції, перетворення Фур’є для конволюції двох послідовностей є результатом множення двох перетворень Фур’є для обох послідовностей. Конволюція (або згортка) є сумою множення елементів одного многочлена, виставлених у звичайному порядку, на елементи іншого, розставлені у зворотньому напрямку:

$$(f \otimes \delta)(x) = \int_{-\infty}^{\infty} f(t)\, g(x-t)\, dt \qquad \textbf{[16]}$$

Відповідно, з цього випливає, що отримавши результат згортки двох послідовностей Фур’є за логарифмічну обчислювальну складність, потрібно перемножити елементи з відповідними індексами один на інший, отримавши згортку цих послідовностей. Після цього за ту ж логарифмічну складність, застосувавши зворотнє швидке перетворення Фур’є, можна обчислити й первинні коефіцієнти результату множення двох рядів, які виявляться результатом множення двох многочленів один на одного. Потрібно лише розмістити елементи умовно другого масиву (насправді неважливо якого саме) в зворотньому порядку перед першою операцією ШПФ, відповідно потім ще раз перевернути вже перед множенням двох перетворень для отримання правильної згортки, з якою наступною операцією зворотнього ШПФ будуть знайдені коефіцієнти множення послідовностей довжиною m + n, де m та n — кількість елементів в першому та другому масивах. **[13]**

Таким чином, розібравши математичний апарат швидкого перетворення Фур’є, наочно переконалися у його корисності для обробки числових рядів та підготовці даних до внесення у процес встановлення збіжностей, який описаний в наступному розділі.

3.3 Алгоритм пошуку з використанням швидкого перетворення Фур'є

Нехай дано дві числові послідовності txt[] та pat[], довжини яких складають відповідно n і m символів певного алфавіту. Потрібно надати кожному символу алфавіту унікальне цілочисельне значення, яке б позначало лише цей символ та слугувало б розрізненням серед всієї множини алфавітних літер. Наприклад, нехай будуть взяті до розгляду літери латинського алфавіту, які будуть означені за номерами в порядку зростання серед символів. Тобто, нехай число 1 позначає літеру "A", число 2 позначає літеру "B", число 3 - літеру "C", а число 4 позначає літеру "D". Відповідно, будь-який текст довільної довжини, де використовуються лише ці 4 символи, будуть перетворені у масив цілих чисел, значеннями від 1 до 4.

Відповідно, можна поелементно порівнювати два числових масиви зі значень, якими заміщені символи, на знаходження збіжності. Наприклад, масив чисел [1 1 2], яким очевидно позначається символьна послідовність (надалі коротко - текст) "AAB", можна порівняти з масивом такої ж довжини "AAC". Числова послідовність для другого текстового рядка, згідно з умовними позначеннями, прийнятими для даного підрозділу, є наступною — [1 1 3]. При поелементному порівнянні використовуються апробація на рівність цілих чисел, які знаходяться в різних масивах, але на однакових індексах в цих масивах. Тобто, зберігаючи нумерацію елементів та пересуваючись на наступний елемент синхронно в обох масивах, порівнюються числа, які знаходяться на однакових місцях у обох масивах. В даному прикладі перевірка, яка почнеться з першого елементу, розміщеного на індексі 0. Оскільки 1 дорівнює одному, робиться висновок, що перші літери співпадають. Синхронно інкрементувавши значення індексу для обох

масивів, можна визначити, що й другий елемент в обох масивах співпадає. А вже на третьому елементі чітко видно, що два елементи в різних масивах є різними. Таким чином, можемо сказати, що перші дві літери співпадають в масивах, але через неспівпадіння третьої літери весь пошук виявився невдалим — збігу між двома текстовими послідовностями не знайдено. Якщо ж порівняємо послідовність "AAB" з послідовністю "AAB", то на всіх трьох індексах, тобто по всій довжині масиву всі елементи співпадають, тому збіг послідовностей зафіксовано. Причому початок збігу, вочевидь, є на елементові, що знаходиться на 0-му індексі першого рядка, якщо сприймати перший рядок як задану незмінну текстову послідовність, а другий рядок як змінну задавану користувачем послідовність для пошуку серед символів першого рядка. Якщо шуканий рядок буде меншим за довжиною від тексту для пошуку, то цикл порівняння завершиться на останньому символі шуканого рядка. Для того, щоб зробити порівняння з текстом повним, необхідно порівняти всі можливі підпослідовності в даному тексті txt[]. Для цього базовий індекс відліку порівняння в тексті інкрементується на 1, після чого цикл посимвольної перевірки повинен повторитися. Повне порівняння буде завершене тоді, коли елемент останнього індекса шуканого зразка pat[m-1] буде порівняний з останнім елементом порівняльного тексту txt[n-1]. Кількість збігів вочевидь збільшується, та може досягти кількості n - m + 1, де n - кількість символів у тексті, m - кількість символів у зразку. Для прикладу, для тексту txt[] = "AAB" = [1 1 2] порівняння з послідовністю pat[] = "AB" = [1 2] почнеться з базового індексу 0. Спершу буде порівнянна підпослідоність txt[0, 1] з шуканим зразком pat, знайдена невідповідність на індексі 1, тому базовий індекс 0 не виявиться точкою збігу. Базовий індекс для наступного циклу збільшиться і стане рівним 1. Підпослідоність txt[1, 2] при порівнянні виявить повну збіжність для обох елепментів, Індекс тексту 1 стане точкою збігу. Оскільки елемент txt[2] є останнім для даної

послідовності, то збільшення базового індекса відліку є недоцільним, так як кількість порівнюваних елементів в шуканому рядку виявиться більшою, ніж в підпослідовності порівнювального тексту, а тому повного збігу символів бути не може. По-іншому, якщо базовий індекс є рівним n - m - 1, то у порівнянні вже немає сенсу. Для порівняння виду txt[] = "AAA" = [1 1 1] з pat[] = "A" = [1] очевидно буде співпадіння по всіх індексах, від 0 до індекса (n - m + 1) - 1.

Переобтяженість даного простого підходу до пошуку співпадінь змушує шукати нові способи порівнянь. Набагато ефективнішим був би спосіб такої обробки даних в числових масивах, що дозволив би за один прохід по тексту відразу віднайти точки співпадіння. Такий спосіб перетворення був знайдений, для його реалізації потрібно віднайти результат множення двох многочленів.

Простий числовий масив містить всі значення символів тексту. Але при цьому всі позиції, індекси числового масиву є рівноцінними, отже не містять інформації про порядок розташування. Для прикладу, масиви [1 2 3] та [3 1 2] містять одні й ті ж елементи, але оскільки їх поява не збігається по індексах для обох масивів, порівняння цих рядків дасть той же результат, що і порівняння рядків, складених з різних числових елементів, - тобто негативний. Тому для пошуку однозначної ознаки відповідності, яка могла б спростити задачу входжуваності одного рядка в інший, потрібно математично розподілити числа так, щоб їхня позиція впливала на кінцевий результат. Таким способом є вираження числової послідовності $[a_0, a_1, a_2...a_{n-1}]$ у вигляді коефіцієнтів деякого многочлена $a_0x^0 + a_1x^1 + a_2x^2 + ... + a_{n-1}x^{n-1}$. Тоді степені коренів многочлена будуть відображати позицію даного елемента первинного масива у математичну площину. Відповідно кожен з доданків многочлена має більшу обчислювальну розкиданість значень, що є значним кроком для

встановленння розрізнюваності не лише для елементів, а й для їхніх позицій в масиві.

Суть методу подальшого пошуку співпадінь, який був знайдений емпірично, базується на наступному твердженні.

Якщо дано два многочлени

$$f(x) = a_0 + a_1x + a_2x^2 + ... + a_{n-1}x^{n-1}$$

та

$$g(x) = b_0 + b_1x + b_2x^2 + ... + b_{m-1}x^{m-1}$$

при n >= m, то результат множення другого многочлена самого на себе

$$g(x) * g(x) = t_0 + t_1x + t_2x^2 + ... + t_{2m-2}x^{2m-2}$$

міститиме коефіцієнт t_{m-1} на індексі m-1, який буде рівний тим і тільки тим коефіцієнтам z_s результату множення двох многочленів

$$f(x) * g(x) = z_0 + z_1x + z_2x^2 + ... + z_{n+m-2}x^{n+m-2}$$

у проміжку m-1 <= s <= n-1, де s буде кінцевим індексом повного співпадіння частини коефіцієнтів у многочлені f(x) з коефіцієнтами многочлена g(x), тобто ряд коефіцієнтів $[a_{s-m+1} ... a_s]$ буде точно збігатися з рядом коефіцієнтів $[b_0 ... b_{m-1}]$.

Говорячи простими словами, дане твердження свідчить про те, що результат множення одного многочлену самого на себе містить елемент, назвемо його ознакою збігу, який може зустрічатися в результаті множення цього многочлена на інший многочлен, більший або такий же за розміром, на тих місцях, де закінчується точне співпадіння коефіцієнтів.

Тут варто зробити важливе зауваження: ознака збігу може зустрічатися в результаті множення двох многочленів на індексах, непридатних для співставлення, але однак вона буде справедливо відкинута через непотрапляння шуканого набору коефіцієнтів многочлена в рамки кількості коефіцієнтів досліджуваного многочлену. Наприклад, якщо ознака збігу присутня до індексу m - 1, то можна вирахувати, що шуканий набір

коефіцієнтів довжиною m - 1 присутній серед коефіцієнтів досліджуваного многочлена, але індекс старту співпадіння є меншим нуля, тобто знаходиться раніше за початок досліджуваного многочлена, а це неможливо.

Тепер потрібно спробувати довести це твердження. Множення многочленів $f(x) = a_0 + a_1x + a_2x^2 + ... + a_{n-1}x^{n-1}$ та $g(x) = b_0 + b_1x + b_2x^2 + ... + b_{m-1}x^{m-1}$ один на одного означає перемноження всіх їх доданків між собою. Згрупуємо новоутворені доданки за однаковими коефіцієнтами першого многочлена, що множаться на члени другого многочлена.

Загальний вираз набуде вигляду:

$$\mathbf{(a_0 + a_1x + a_2x^2 + ... + a_{n-1}x^{n-1}) * (b_0 + b_1x + b_2x^2 + ... + b_{m-1}x^{m-1}) = (a_0b_0x^0 + a_0b_1x^1 + ... + a_0b_{m-1}x^{m-1}) + (a_1b_0x^1 + a_1b_1x^2 + ... + a_1b_{m-1}x^m) + ... + (a_{n-m-1}b_0x^{n-m-1} + a_0b_1x^1 + ... + a_0b_{m-1}x^{m-1})}.$$

Звідси бачимо, що всі коефіцієнти обох многочленів перемножуються один на одного, а корінь многочлена змінює степінь в два цикли. Можна представити ряд кореня у відповідно згрупованому результаті множення многочлена у вигляді x^{t+d}, де t це число від 0 до n-1, та є показником зовнішнього циклу, а d змінюється від 0 до m-1 для кожного значення t, повторюючи значення d у внутрішньому циклі:

[0, 1, 2, ..., m-1, 1, 2, 3, ... , m, 2, 3, 4, ..., m+1, ... , n-m-1, n-m, n-m+1, ... , n-1].

Отримавши графік зміни степенів в даній розкладці множення многочлена, визначимо частоту появи всіх степенів кореня. Виявиться, що степені 0 та n-1 зустрічаються біля коренів лише раз, степені 1 та n-2 зустрічаються серед доданків 2 рази і так далі, по мірі проходження дистанції в довжину другого многочлена. Починаючи з степеня m, цей та наступні показники степенів до n-1 будуть зустрічатися серед доданків m разів. На практиці це означає, що на формування ряду доданків з коренем x^f, де f є величина між m та n-1, як уже було сказано, впливатиме розстановка всіх

коефіцієнтів всередині першого многочлена, що знаходяться між індексами f-m та f. Більше того, можна переконатися, що коефіцієнти множаться біля одностепеневого кореня максимальної частоти серед доданків многочлена наступним чином:

$$a_{f-m}b_{m-1}x^f + a_{f-m+1}b_{m-2}x^f + \ldots + a_f b_0 x^f$$

Тобто розкладка множення коефіцієнтів є нічим іншим як конволюцією другого многочлена з частиною першого такої ж довжини. Значення цієї конволюції матимуть унікальні відмінні значення навіть у випадку присутності одних і тих же коефіцієнтів, але розміщених в іншому порядку.

Якщо в першому многочлені трапляється повне співпадіння коефіцієнтів з другим многочленом, то на індексі останнього співпадаючого елементу в новому ряді коефіцієнтів результату множення цих многочленів знаходитиметься унікальне значення, що може повторитися на іншому індексі лише у разі нового співпадіння частини першого многочлена з другим. Це значення буде ознакою збігу другого многочлена з першим за індексом, що розташований в першому многочлені на довжину другого многочлена раніше від індексу в результуючому многочлені, на якому знайдене співпадіння коефіцієнтів.

Варто ще раз підкреслити, що дане порівняння коефіцієнтів результуючого многочлена з ознакою збігу потрібно здійснювати лише на тих повноцінно сформованих значеннях, в обрахуванні яких брали участь всі коефіцієнти другого многочлена: тобто стартуючи з індекса m-1 до індекса n-1.

Таким чином, пошук символьного рядка в числовому вигляді у фрагменті досліджуваного тексту може бути виконаний шляхом двох множень многочленів. Спершу множиться другій многочлен сам на себе, знаходиться опорний коефіцієнт, тобто ознака, збігу. Далі виконується множення досліджуваного тексту на шуканий рядок, і в результуючому

многочлені робиться один прохід розміром в довжину тексту з порівнянням його коефіцієнтів з ознакою збігу. У випадку збігу на індексі f результуючого многочлена, індекс співпадіння в тексті вичисляється як f-m.

Оскільки множення многочленів за допомогою методу швидкого перетворення Фур’є можна виконати за обчислювальну складність O(nlogn), то загальна складність такого алгоритму становить O(2nlogn + n).

4. Програмна реалізація методу пошуку з використанням швидкого перетворення Фур’є

4.1 Архітектура

Для розробки була використана мова Сі++. Засоби об’єктно-орієнтованого програмування не були використані, так як предметна область в реалізації обмежувалася двома вхідними масивами з двох файлів. Створення функціоналу з допомогою використання класів не було передбачено. Алгоритм є по факту послідовною обробкою цих масивів та виведення результату у вигляді масиву числових значень індексів співпадінь. Для реалізації математичних можливостей, зокрема для використання комплексних чисел, була використана стороння бібліотека `<complex>`. Також було використано класи стандартної бібліотеки такі як `string` та `vector` для швидкого визначення довжини масиву, символьного чи числового, а також для полегшення додавання та прибирання певних елементів, для реалізації чого засобами, для прикладу, мови Сі, знадобилися б додаткові функції створеннння та роботи з динамічним масивом.

Основна функція програми включає в себе початкову фазу входження символьних даних тексту та рядка в програму та їх приведення в належний для роботи основного алгоритму вигляд, та власне основна частина, - робота алгоритму пошуку співпадінь тексту та рядка та виведення у вигляді масиву індексів. Робота алгоритму обрамлена функціями, які допомагають зняти часові мітки для визначення грубого часу роботи даного алгоритму.

Після виконання пошуку нашим алгоритмом, в основну функцію додана для наочного порівняння функція пошуку рядка на основі алгоритму Рабіна-

Карпа. Часові мітки навколо неї також допомагають визначити час виконання роботи алгоритму.

Діаграма прецедентів для даної програми представлена в розширеному вигляді. Деякі функціональні можливості є запропонованими для можливого внесення в проект. Наразі в них немає потреби, так як є предметом роботи доведення запропонованого алгоритму до конкурентної дії.

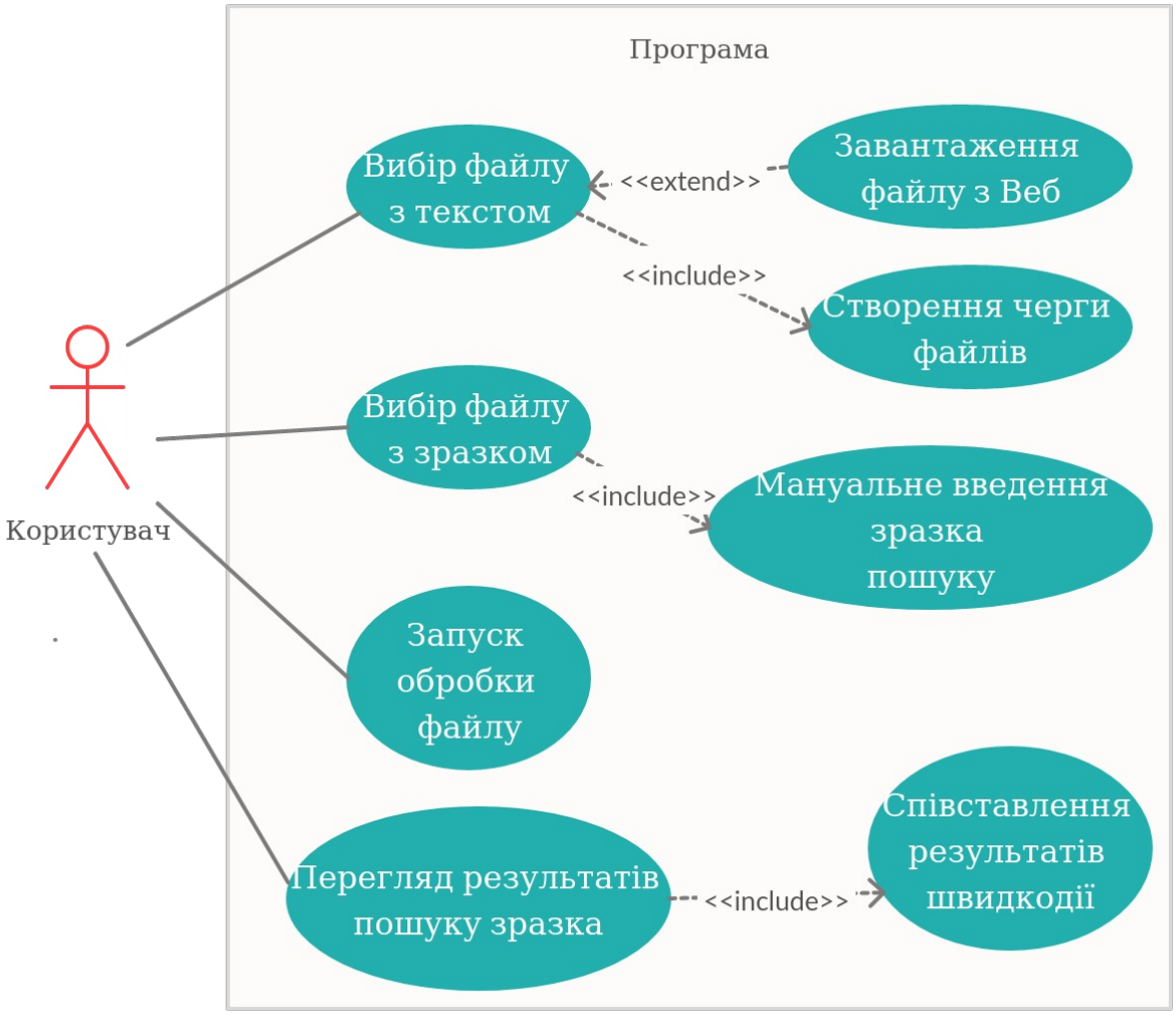


Рисунок 4.1 - Діаграма прецедентів пошуку підрядка

Для кращого розуміння ходу програми була також розроблена діаграма послідовностей. Програма не є ні багатокомпонентною, ані розподіленою. Тим не менше, в ній є кілька важливих субмодулів, які заслуговують на наочну демонстрацію. В якості інтерфейсу використовується вікно терміналу ОС Ubuntu, для використання на якій і призначена ця програма. Обробник тексту включає в себе модуль пошуку файлу в файловій системі комп’ютера, створення потоку з файлу в буфер пам’яті у вигляді символьного масиву, прийом текстового зразка на обробку, а потім переведення чисел-еквівалентів з ASCII-таблиці обох масивів символів у інший набір чисел, який буде легше оброблятися в останньому субмодулі - власне пошукові збігів в текстовому масиві. Останній субмодуль включає в себе алгоритм пошуку та масив з результуючими індексами. Їх, після завершення роботи алгоритму, буде відправлено для відображення користувачу з допомогою інтерфейсу, а саме в консольному вікні, де була запущена програма. Після цього програма завершить свою роботу.

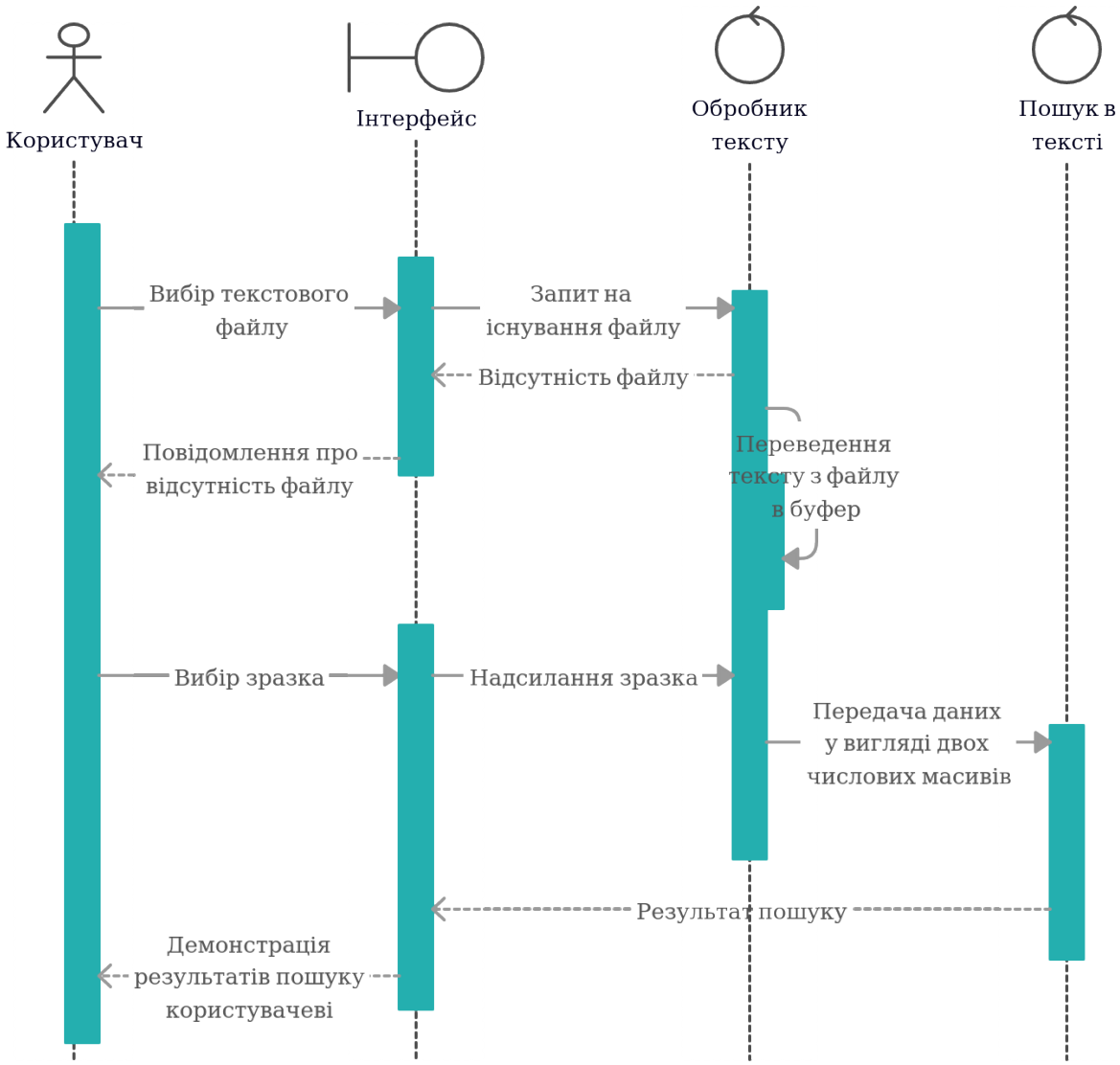


Рисунок 4.2 - Діаграма послідовностей пошуку підрядка

4.2 Структури даних

Для даного проекту об’єктно-орієнтований підхід використаний не був. Як вже було сказано, із специфічних структур представлення даних були використані тільки шаблонні класи `string` та `vector`. Головною їх перевагою порівняно з використанням звичайних масивів є можливість використовувати методи додавання та прибирання елементів з масиву. Ці методи корисні у випадку динамічного призначення та зміни розмірів цих масивів. Дана програма не потребує додаткових класів, так як не описує велику кількість взаємозалежних сутностей матеріального світу, а реалізує послідовність дій, в ході яких певний набір даних зазнає перетворення, та знаходиться результат, який демонструється перед завершенням програми.

Тим не менше, уявивши цю програму розробленою в середовищі виключно об’єктно-орієнтованої мови програмування, можна приблизно уявити, які структури даних були б використані для вирішення задачі пошуку в тексті. В першу чергу, потрібен клас графічного інтерфейсу, для обміну повідомленнями між користувачем та програмним продуктом, як-то введення тексту пошуку в консоль терміналу та виведення кінцевої інформації про статус перебігу пошуку, результати пошуку збігів

Як найважливіший компонент можна сформувати клас з умовною назвою Текстовий Масив. В цьому класі було б сформоване обов’язкове поле масиву символьних даних. Воно могло б бути ініціалізованим при створенні об’єкта даного класу шляхом запису в конструкторі класу послідовності входження символів з файлу або з буферу вхідної-вихідної пам’яті (у разі режиму введення даних з клавіатури) в поле масиву символів. В цей же клас міг би ввійти числовий масив динамічного розміру. З допомогою

внутрішнього методу реалізувати переведення символів у числа згідно з концепцією знаходження збігів з використанням методу ШПФ.

Доповнити цю модель до загальної картини потрібно додаванням класу контролера. В його основний публічний метод потрібно помістити загальний набір функцій, який виконує перетворення Фур'є для двох числових масивів, що передадуться в цей метод в якості аргументів. Ці функції, які виконують різні етапи перетворення даних можуть існувати як методи всередині цього ж класу. Але пропонувалося б зробити їх приватними, не відкритими, так як їх використання обмежене лише роботою даного функціоналу, а використання їх як публічних могло б створити небезпечну ситуацію введення неперевірених даних, чий розмір не був би рівним експоненціальній функції числа 2.

Таким чином, загальна модель розбиття на класи може бути застосована для об'єктно-орієнтованого підходу, але в нашому випадку її застосування лише ускладнило б і невиправдано збільшило б програму.

4.3 Основні функції

Прийом даних у початковій фазі роботи алгоритму Фур'є відбувається зчитуванням тексту та рядка з двох різних файлів у окремі символьні масиви. Початковою думкою було зробити отримання тексту програмою саме з файлу, але отримання рядка дозволити напряму від користувача для більшої зручності та швидкості роботи з програмою. Однак середовище розробки `Sublime` не дозволяє отримання введення інформації напряму від користувача з допомогою функціоналу перевантаженого класу `std::cin`. До того ж, як вже було описано вище, ідея отримання вхідної інформації з

текстового файлу є більш прийнятною для використання з великими за обсягом текстовими зразками. Конкретна функція, що використовується в цих цілях, має вигляд `int getLineFromFile(string& fileName, string& textLine, vector<int>& v_text)`. Вона приймає назву файлу, де знаходиться символьна інформація, що мусить бути оброблена, посилання на об'єкт типу string (символьний масив), в який буде закинуто посимвольно весь текст, що знаходиться у файлі. Далі виконується функція `translateToAscii` переведення всіх символів у числове відображення з завантаженням всіх отриманих чисел в числовий вектор `v_text`, посилання на який передається в аргументах обох функцій. В разі відсутності файлу з текстом на екран буде виведено відповідне повідомлення, функція поверне результат своєї роботи 1, після чого головна функція теж припинить свою роботу з тим самим результатом помилки 1. Функція `translateToAscii` виконує переведення символів латинського алфавіту без спеціальних розширених літер, в числа, за наступним сценарієм: великі та малі літери стають числами, 'A' або 'a' стає 2, 'B' або 'b' стає 3 ітд, Неалфавітні символи, цифри, - зберігають свої відповідні ASCII значення. Керуючий символ нового рядка `'\n'`, чиє значення в таблиці дорівнює 10, задля незаплутаності приймає значення на 128 більше від свого номіналу, тобто стає 138. Таким чином, йде перекидання у значення другого напівбайту, де розміщуються символи Розширеної ASCII таблиці, які не враховуються при нашому переведенні символів. Знак **'*',** поставлений не на початку і не в кінці тексту, розцінюється як підстановка будь-якого іншого символу з таблиці. В вектор цілих чисел даний символ заноситься як 0. При виконанні швидкого перетворення Фур'є це значення зіграє роль встановлення збіжності тексту та рядка, який би символ не порівнювався з **'*'.** Щоправда, дана підстановка не діє в нашому алгоритмі, якщо поставити даний символ в самому початку або кінці тексту, можлива лише заміна букви в середині певного тексту.

Після того, як функція перекладу тексту в числа двічі відпрацювала на файлах, в програмі є два заповнені числові вектори, - для тексту і для шуканого умовного рядка. Посилання на ці два вектори потрапляють у функцію алгоритму `int fourierComparison(vector<int>& v_text, vector<int>& v_pattern, vector<int>& match_indices)`. В якості аргументу в цю функцію потрапляє також попередньо створений вектор, що міститиме числа індексів співпадінь, тобто потрібний нам результат виконання програми. Повертане значення цією функцією вказує на помилковість чи успішність виконаних над даними маніпуляцій. Зокрема, враховані наступні можливі помилки, при яких функція повертає код помилки — оброблюваний текст або шуканий рядок відсутні (кількість символів дорівнює нулю), шуканий рядок довший за текст, в якому пропонується провести пошук даного рядка.

Скорочена суть даного методу імплементації швидкого перетворення Фур'є, реалізованого в цій функції, полягає в наступному: спершу потрібно знайти коефіцієнти многочлена, утвореного множенням двох однакових многочленів з чисел шуканого рядка, отримавши ряд повної збіжності двох рядків, останній коефіцієнт в результуючому многочленові буде унікальним ідентифікатором, - опорний визначник збіжності — який засвідчує збіжність двох рядків певної однакової довжини. Після цього потрібно повторити операцію множення двох многочленів, але на цей раз множниками вже мають бути ряд чисел перетвореного шуканого рядка та ряд чисел перетвореного оброблюваного тексту. Отриманий многочлен потрібно пройти у пошуку опорного коефіцієнта, індекс, де він буде зафіксований, буде останнім символом у конкретно взятому підрядку тексту, тому для визначення конкретного значення індексу початку збіжності двох рядків потрібно від значення індексу збіжності опорного коефіцієнту просто відняти довжину шуканого рядка.

Швидке перетворення Фур’є трапляється в нагоді для пришвидшеного множення двох многочленів. Але для застосування даної функції спершу треба зробити деякі перетворення над самими числовими послідовностями, зокрема, потрібно з допомогою додавання нулів (як невпливаючих на кінцевий результат чисел) збільшити величину векторів чисел шуканого рядка та тексту до розміру, в який може ввійти добуток цих двох многочленів, привести розмір тексту та рядка у відповідність з задоволенням експоненціальної функції з основою 2, так як швидке перетворення Фур’є базується на алгоритмові типу *divide and conquer*, що являє собою застосування операції рекурсивного поділу шуканого масиву значень на 2 половини з застосуванням певного комплексу операцій.

Основні етапи функції `fourierComparison` наступні:

1) Нормалізація векторів, - приведення їх до відповідних розмірів;

2) Множення Фур‘є числового ряду шуканого рядка на себе самого для пошуку ряду збіжності та опорного коефіцієнту;

3) Множення Фур‘є числового ряду тексту на числовий ряд рядка для пошуку ряду порівняльного співставлення;

4) Зменшення ряду порівняльного співставлення до початкового розміру тексту шляхом відкидання його “хвоста”;

5) Пошук в ряду порівняльного співставлення опорного коефіцієнта та ,у випадку його знаходження, - знаходження індекса початку повного співпадіння частини тексту та рядка.

Функція нормалізації векторів приймає посилання на два цілочислові вектори. Дана функція виводить значення суми кількості елементів двох векторів. Після чого отримане число співставляється та в разі потреби підганяється до двійки у певній степені для задоволення умови про складність `n log n`. Вектори тексту та рядка збільшуються до розміру цього знайденого числа, на вакантні місця ставляться нулі.

Далі застосовується функція, в якій власне й реалізоване множення многочленів з допомогою швидкого перетворення Фур'є `fourierMultiplying(vector<int>& v_text, vector<int>& v_pattern, vector<int>& v_result, bool self_compare)`. Функція приймає як аргументи посилання на два порівнювані вектори, - тексту та шуканої послідовності, а також посилання на результуючий вектор, та булеве значення, чиє позитивне значення інтерпретується як необхідність виконати дану функцію лише з одним вектором, помноживши його самого на себе. Цей варіант функції якраз йде першим у функції `fourierComparison`, так як многочлен рядка множиться сам на себе для знаходження послідовності точної відповідності. Ця функція потребує додаткової деталізації.

Для множення многочленів потрібно знайти ряд коренів обох многочленів для всіх комплексних коренів з одиниці (дискретне або швидке перетворення Фур'є), потім почергово перемножити відповідні коефіцієнти в отриманих рядах Фур'є обох многочленів, а до отриманого ряду чисел застосувати зворотне перетворення Фур'є, отримавши за теоремою про конволюцію коефіцієнти множення многочленів у швидший спосіб, ніж перемножувати їх один на одного напряму.

Швидке перетворення Фур'є реалізовується шляхом рекурсивного поділу набору коефіцієнтів на парні та непарні частини, які додаються та віднімаються одна від одної за законом метелика $a + b*e^{2*PI*i*k*n/N}$ та $a - b*e^{2*PI*i*k*n/N}$, де `e`, `PI` та `i` — математичні константи, `a` та `b` — коефіцієнти многочлена, результат додавання та віднімання потім заміщує попередні значення цих коефіцієнтів, `N` є кількістю оброблюваних зразків загального масиву за одну ітерацію або рівень рекурсії, `n` - це індекс конкретного зразка в межах, заданих загальною кількістю, `k` — прогонний індекс коефіцієнтів всередині половини кількості оброблюваних зразків `N`, до коефіцієнтів на

цьому індексі та на такому ж індексі з іншої половини оброблюваних зразків буде застосований закон метелика для визначення нових значень коефіцієнтів на цих індексах. Таким чином, набір змінних `k*n/N` є визначальним для певного рівня рекурсивного поділу або ітерації, якщо застосовано метод динамічного програмування, він визначає комплексний коефіцієнт при інтеракції між коефіцієнтами, розміщеними на певних індексах. Визначення цього коефіцієнта доцільно проводити з врахуванням двох змінних, - кількості зразків в даному циклі та індекс коефіцієнта, що є показником зсуву, чим забезпечується охоплення всього масиву.

Швидке перетворення Фур'є базується на роботі з комплексними числами. Саме тому першим етапом виконання функції `fourierMultiplying` є переведення цілочисельних векторів-відображень тексту та рядка у вектори комплексних чисел за принципом ціле число стає реальною частиною комплексного числа, а його уявна частина ініціалізується нулем.

Оскільки обрахунки починаються після повного рекурсивного занурення, то перші операції відбуваються не на сусідньо розташованих індексах. Індекси елементів, які першими обчислюються, формуються за принципом поділу навпіл за парно-непарною системою. Для даної реалізації було обрано не рекурсивний алгоритм, а ітеративний, для зменшення кількості операцій обчислювальних операцій. Для такого підходу коефіцієнти многочлена потрібно відразу розташувати в порядку їх первинної взаємодії, і тоді на наступних ітераціях розширюючи кількість зразків, що обробляються, обчислювані коефіцієнти будуть автоматично розташовані на правильних місцях. Індекси елементів парно-непарного поділу формують таблицю, яка може бути сформована також операцією реверсу бітів в бінарному відображенні чисел, які є звичайними інкрементованими індексами від 0 до кінця масиву. Головним для даної операції є обчислення старшого біта, який

стане нульовим при реверсній перестановці, ну й, відповідно, нульовий біт первинного індекса повинен стати на місце старшого біта. Старший біт визначається як перший зверху біт, рівний одиниці, у найбільшому числі масиву. Масив, як уже було зазначено, складається з інкрементованих індексів, тобто його значення є індексно-еквівалентними. Отже, останній елемент масиву і буде, власне, старшим числом, з присутністю найстаршої одиниці, так як кількість елементів на цьому етапі виконання програми буде рівним 2 в певному степені.

Для цієї операції застосовується функція `bit_reversal(vector<complex<double> >& vect, unsigned int n)`. Всередині функції створюється тимчасовий масив, котрий ініціалізується значеннями з переданого за посиланням як аргумент в дану функцію вектора. Масив, переданий як аргумент, буде переініціалізований значеннями з тимчасового масиву відповідно до новоствореної в цій функції індексації. Для переіндексації використовується цілочисельний масив, який по порядку буде ініціалізовуватися значеннями індексів, коректними для виконання швидкого перетворення. Пошук старшого біта, описаний вище, здійснюється стартуючи з 31-го біта, найбільшого для типу unsigned int, і продовжується сповзанням вниз, допоки не буде знайдено старший біт числа кількості елементів у масиві. Індекс старшого біта показує загальну кількість бітів, що мусять бути переставлені. Старші біти не підлягають перестановці. Реверс здійснюється установкою молодшого біта результуючого індекса в нуль або одиницю відповідно до значення старшого біта первинного індексу, після чого відбувається зсув бітів результуючого індекса на 1 вправо, а первинного індексу вліво, і продовження установок допоки первинний індекс не стане рівним нулю. Наприклад, в межах 8-ми елементів старшим елементом буде 0x08, або 0b00001000. Старший біт — 3. На нульовому індексі число 0 == 0b0000. Після “перевертання” воно так і залишиться

нулем. На 1-му індексі число 1 == 0b0001. Після реверсу воно буде рівним 0b0100, тобто 4 ітд. Після чого в результуючий масив сусідніми потраплять ті значення, які першими взаємодіятимуть при роботі швидкого перетворення.

Після того, як коефіцієнти розташовуються у вигідному для обчислення порядку, йде переобчислення коефіцієнтів. У функції `fourierTransform` для швидкого перетворення коефіцієнтів застосовуються 3 вкладені цикли. На нижчому щаблі вкладеності обчислюються нові значення коефіцієнтів за законом метелика $a = a + b*w^k; b = a - b*w^k$, просування в циклі здійснюється, щоб забезпечити охоплюваність всіх коефіцієнтів повного масиву. На другому рівні вкладеності відбувається домноження комплексного коефіцієнту `w` на самого себе, так як коефіцієнт зсуву всередині зразка масиву `k` зростає на 1 по мірі просування по індексу всередині зразка, тобто `w^0` стає `w^1`, потім `w^2` ітд. Верхній цикл грає роль визначення розміру зразка в масиві, тобто широту розташування взаємодіючих елементів масиву для даної ітерації. На початку роботи алгоритму взаємодіють два сусідні елементи в заново вибудованому через реверс бітів масиві, наприклад, для масиву з 8 елементів порядок першої половини буде 0, 4, 2, 6; взаємодіятимуть сусідні 0 і 4, потім 2 і 6 індекси ітд, комплексний коефіцієнт при цьому `w^0`, потім розмір зразка зростає в два рази і взаємодіють два елементи через один: 0 і 2, а елемент між ними, - 4 — взаємодіє з елементом 6 уже з коефіцієнтом `w^1`, так як розмір зразка виріс і став рівним 4, тому індекс 0 в 0-ій позиції, а індекс 4 знаходиться на 1-ій позиції відносно умовної осі, що поділяє зразок розміром 4. Таким чином, степінь коефіцієнта зсуву `w` виріс на одиницю. Цей цикл триває допоки розмір зразка не стає рівним розміру всього масиву. Окрім цього, розмір зразка `N` впливає і на первинне визначення самого коефіцієнту $w == e^{2*PI*i*k/N}$. Тому на кожній ітерації верхнього циклу йде перевизначення цього коефіцієнту. Спочатку `N` замінюється на в 2 рази меншу

змінну `bind_step`, щоб виключити `2` з чисельника виразу степені. $e^{PI*i*k/bind\_step}$ зручно обчислити як `cos(PI/bind_step) – i*sin(PI/bind_step)`. Стартове присвоєння цій змінній для 0-ої позиції, з якої почин.ається кожна ітерація верхнього обрамляючого циклу, буде `1`, так як `w^0 == 1.` Далі з допомогою домноження на комплексний коефіцієнт йде просування по зсуву всередині зразка в другому вкладеному циклі, і відповідно зростання цього коефіцієнту з кожною ітерацією (наприклад, `w^1, w^2, w^3` при розмірі зразка 8).

Таким чином, вектор комплексних чисел поступово заповнюється значеннями, які впливають на подальше перевизначення їх самих. По завершенню роботи всіх циклів вектор буде містити ряд Фур’є, тобто корені многочлена для всіх можливих для цього многочлена коренів з одиниці.

Повертаючись до функції `fourierMultiplying`, потрібно сказати, що з допомогою попередньо розібраної функції `fourierTransform` мі знайшли всього один ряд Фур’є для одного вектора. Якщо завдання полягає в тому, щоб знайти ряд повної збіжності для вектора текстового зразка, то однієї операції знаходження ряду Фур’є буде достатньо. По булевому значенню “істина”, що позначає операцію співставлення вектора зі своїм еквівалентом, для всіх елементів вектора буде здійснена операція множення цих елементів самих на себе. Якщо ж ця функція застосовується для знаходження результату множення двох різних многочленів, то негативне булеве значення дасть хід іншій гілці виконання даної функції — спочатку буде отримано два різні ряди Фур’є з двох векторів, а потім вони поелементно перемножаться.

Наостанок до отриманого ряду чисел множення рядів Фур’є, для того, щоб ми отримали ряд коефіцієнтів множення початкових многочленів, згідно з теоремою конволюції, потрібно застосувати операцію зворотного швидкого перетворення Фур’є.

Ціль даної операції полягає якраз у знаходження первинних коефіцієнтів многочлена по ряду Фур'є. Її суть — в заміні степеней комплексних коефіцієнтів `e` на мінусові, та діленню кожного елемента на загальну кількість елементів масиву. Дана операція може здійснюватися кількома швидкими способами, для програми був вибраний один з них, і втілений у функції `void inverseFFT(vector<complex<double> >& vc)`. Її етапи роботи наступні: вектор комплексних чисел (нагадаю, це результуючий вектор множення двох рядів Фур'є), потрапляє в функцію за посиланням, до кожного елемента цього вектора застосовується бібліотечна Сі++ функція кон'югації, тобто зміна знаку уявної частини комплексного числа на протилежний, отриманий вектор перетворюється на ряд Фур'є з використанням раніше описаної функції `fourierTransform`. Далі на отриманий вектор чекає ще одна кон'югація його елементів. Після цього здійснюється ділення кожного елемента масиву на значення кількості цих елементів у масиві.

Сформований вектор після роботи функції `inverseFFT` потрібно позбавити останнього елементу, який завжди є зайвим нулем, та ще раз прогнати всі його елементи для відсіювання та округлення дійсних частин комплексних значень елементів масиву, та запису цих значень у вектор цілих чисел.

В більш загальній функції `fourierComparison` функція `fourierMultiplying` застосовується двічі — і для самого шуканого зразка з булевим аргументом самопорівняння "істина", і для шуканого зразка з оброблюваним текстом з булевим аргументом самопорівняння "неправда". Після знаходження коефіцієнтів множення многочленів тексту та зразка, залишається перша частина коефіцієнтів довжиною в оброблюваний текст, слідуюча частина відкидається. Серед коефіцієнтів ряду збіжності, тобто вектора множення многочлена шуканого зразка самого на себе, для

подальшого пошуку збіжності нас цікавить лише один, останній у векторі, елемент — опорний визначник збіжності. Це значення єдиноможливо зустрічається як коефіцієнт множення двох многочленів для певної кількості елементів лише у випадку входження одного многочлена в інший. Причому його поява вказує на хвіст даної збіжності. Таким чином, в функції `fourierComparison` останнім пунктом йде проходження по всіх елементах коефіцієнтів множення тексту та зразка, з порівнянням кожного елемента з опорним визначником збіжності. В разі співпадіння цих чисел, від індекса співпадіння відмінусовується кількість символів у шуканому зразку, і таким чином знаходиться індекс початку точного співпадіння частини тексту та зразка, та заноситься в масив індексів співпадінь.

5. Тестові випробування

5.1 Керівництво користувача

Програма являє собою виконуваний в операційній системі Ubuntu об’єктний файл типу **.o**. Запускається дана програма у вікні терміналу, туди ж вона виводить всі результати своєї роботи.

Користувачу пропонується вибрати файл з присутніх у файловій системі комп’ютера в якості тексту, в якому він може провести пошук текстового зразка. В іншому режимі програма сама відкриє файл з певною назвою в основній директорії користувача, який за замовчуванням повинен містити в собі досліджуваний текст. Для різного розміру зразків існуватимуть 2 різні підходи введення пошукового зразка, користувач зможе вводити шукану послідовність з клавіатури прямо в консольне вікно, або ж вибирати файл для відкриття з списку. Наразі файл із шуканим зразком вибирається програмою автоматично, за замовчуванням, шукаючи відповідний за назвою файл.

Після запуску програма відпрацьовує та висвітлює результати свого виконання в консолі, показуючи кількість символів у тексті та зразкові, результати пошуку у вигляді числових значень номерів у тексті, де зустрічаються співпадіння із зразком. Окрім цього, на екран будуть виведені аналогічні результати пошуку для порівняння, проведені іншим алгоритмом пошуку зразка в тексті, а саме алгоритмом Рабіна-Карпа. Також для наочного співставлення швидкодії двох алгоритмів в термінал буде виведено час виконання цих алгоритмів.

Програма після запуску відпрацьовує лише раз, та припиняє роботу. Після змін у файлах зразка та досліджуваного тексту, програма для проведення повторного пошуку повинна бути запущена повторно.

При запуску без аргументів програма спробує знайти та відкрити текстові файли з зразком та основним текстом. Якщо ж запускати з аргументами, тобто перетягнути в вікно терміналу після введеної назви програми спочатку файл з текстом, а потім файл зі зразком (повні шляхи у файловій системі), то програма відкриє та використає для пошуку саме ці два файли.

Програма призначається в першу чергу розробникам, які можуть зацікавитись можливостями удосконалення та конкуретною спроможністю алгоритму пошуку на основі ШПФ.

5.2 Результати тестування пошуку з використанням швидкого перетворення Фур'є

Фінальна швидкість алгоритму є, на даний момент, повільною для звичайних задач пошуку по зразку.

Приклади швидкодії для різних видів пошуку:

1) Повне неспівпадіння малого зразка

Довжина тексту - 4382 знаків

Довжина рядка - 1 знак

Рабін-Карп - 78 мікросекунд

Фур'є - 22781 мікросекунд

2) Повне співпадіння малого зразка

Довжина тексту - 4382 знаків

Довжина рядка - 1 знак

Рабін-Карп - 77 мікросекунд

Фур’є - 28548 мікросекунд

3) Повне співпадіння великого зразка

Довжина тексту - 4382 знаків

Довжина рядка - 4382 знаків

Рабін-Карп - 118 мікросекунд

Фур’є - 50303 мікросекунд

4) Повне неспівпадіння великого зразка

Довжина тексту - 4382 знаків

Довжина рядка - 4382 знаків

Рабін-Карп - 114 мікросекунд

Фур’є - 47535 мікросекунд

Похибка точності — 10 мікросекунд для Рабіна-Карпа, 1000 мікросекунд для ШПФ.

Складність алгоритму за найгіршим сценарієм (n — довжина тексту, m — довжина рядка):

Big O = n +m + 2*(3nlogn + 2(n + m) + n) + n - m = 2*(3nlogn + 4n + m)

Висновки

В результаті роботи над завданням були вивчені класичні алгоритми пошуку зразка, як числові так і нечислові. Був знайдений спосіб зручного переведення текстової у числову інформацію. Надалі зроблене припущення стосовно знаходження взаємозв’язку між точками співпадіння тексту та зразка за результатами конволюції двох многочленів. Множення многочленів здійснене з допомогою швидкого перетвореннння Фур’є, що є значно коротшим шляхом до знаходження результату. Останнім пунктом на шляху реалізації програми було знаходження та доведення опорного значення збігу для однакових многочленів, яке точно сходиться при співпадінні зразка та частини будь-якого тексту, та з допомогою якого стає можливим визначити місце співпадіння.

Даний алгоритм в його нинішній реалізації є неконкурентним, так як його швидкодія все ж значно гірша за класичні алгоритми пошуку. Однак дана імплементація є кроком вперед порівняно з іншими підходами, так як фінальне порівняння на збіжність рядів виконується за один прохід, без необхідності повторних перепроходів для довизначення збіжності. Таким чином, цей алгоритм наближається до вирішення задачі визначення збіжності рядів за якимось певним одним значенням, пошуки в цьому напрямі вже довгий час як не можуть увінчатися успіхом.

Список літератури

1. Гасфілд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Ден Гасфілд. – СПб: БХВ-Петербург, 2003. – 513 с. – (БХВ-Петербург). — Сторінка 103

2. Walse K.H. and Ali M.S. Performance analysis of exact pattern matching algorithms [Електронний ресурс] // Advances in Computer Vision and Information Technology. – 2008. – Режим доступу до ресурсу: https://books.google.com.ua/books?id=pNKxKYHL2RYC&pg=PA1259&lpg=PA1259&dq=string+searching+linguistics+large+text+pattern+search&source=bl&ots=mQvJI4rg9j&sig=ACfU3U0hNLTNgnt1WjyC4d7Idq0FCzbZzw&hl=uk&sa=X&ved=2ahUKEwj7xMfM5cnpAhUQmYsKHdIJBB8Q6AEwAHoECAkQAQ#v=onepage&q=string%20searching%20linguistics%20large%20text%20pattern%20search&f=false

3. Pro Git. 2nd edition [Електронний ресурс] // Git-Scm – Режим доступу до ресурсу: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

4. Naive algorithm for Pattern Searching [Електронний ресурс] // Geeks For Geeks – Режим доступу до ресурсу: https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/

5. ICS 161: Design and Analysis of Algorithms [Електронний ресурс] // Eppstein David - Design and Analysis of Algorithms – Режим доступу до ресурсу: https://www.ics.uci.edu/~eppstein/161/960227.html

6. Boyer Moore Algorithm for Pattern Searching [Електронний ресурс] // Geeks For Geeks – Режим доступу до ресурсу: https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

7. String Searching [Електронний ресурс] // Princeton University - COS 226 - Algorithms and Data Structures. – 2004. – Режим доступу до ресурсу: https://www.cs.princeton.edu/courses/archive/fall04/cos226/lectures/ string.4up.pdf.

8. Algorithms in action - FFT [Електронний ресурс] // Lectures by Uri Zwick and Haim Kaplan – Режим доступу до ресурсу: http://www.arazim-project.com/sites/default/files/public/lesson_sums/1fft.pdf

9. Approximate pattern matching [Електронний ресурс] // Max Planck Institut Informatik. – 2013. – Режим доступу до ресурсу: http://resources.mpi-inf.mpg.de/departments/d1/teaching/ss13/strings/slides8.pdf

10. Peter Clifford and Raphael Clifford - Simple Deterministic Wildcard Matching [Електронний ресурс] // University of Bristol, Dept. of Computer Science. – 2006. – Режим доступу до ресурсу: http://citeseerx.ist.psu.edu/viewdoc/download? doi=10.1.1.106.5266&rep=rep1&type=pdf

11. Fast FourierTransforms [Електронний ресурс] // Jeff Ericksson, Lectures on University of Illinois. – 2018. – Режим доступу до ресурсу: http://jeffe.cs.illinois.edu/teaching/algorithms/notes/A-fft.pdf.

12. Fast Fourier Transform [Електронний ресурс] // CS 6505, Computability and Algorithms - Lecturer: Santosh Vempala. – 2010. – Режим доступу до ресурсу: https://www.cc.gatech.edu/classes/AY2010/cs6505_spring/lectures/FFT.pdf.

13. Polynomial Multiplication and Fast Fourier Transform [Електронний ресурс] // Iowa State University, Department of Computer Science - Yan-Bin Jia. – 2019. – Режим доступу до ресурсу: http://web.cs.iastate.edu/~cs577/handouts/polymultiply.pdf.

14. Lecture 11 Fast Fourier Transform (FFT) [Електронний ресурс] // Hong Kong Baptist College - Weinan E, Tiejun Li – Режим доступу до ресурсу: http://www.math.hkbu.edu.hk/~zeng/Teaching/math3620/lect1E-fft.pdf

15. An Interactive Guide To The Fourier Transform [Електронний ресурс] // BetterExplained - Kalid Azad – Режим доступу до ресурсу: https://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/

16. Fourier Transforms and theFast Fourier Transform (FFT) Algorithm [Електронний ресурс] // Carnegie Mellon University - Notes 3, Computer Graphics 2, 15-463. – 1998. – Режим доступу до ресурсу: http://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/2001/pub/www/notes/fourier/fourier.pdf

17. Smith S. W. The Scientist & Engineer's Guide to Digital Signal Processing / Steven W. Smith., 1997. – (1). – (ISBN 0-9660176-3-3).

Додаток 1.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

Специфікація

роботи на тему

“ШВИДКЕ ПЕРЕТВОРЕННЯ ФУР’Є В ЗАДАЧАХ ОБРОБКИ ТЕКСТІВ”

Виконав
студент групи ТВ-з61
кафедри АПЕПС
факультету ТЕФ
Джулай Володимир

Київ - 2020

Таблиця 2. Специфікація

Позначення	Найменування	Примітки
Документація		
1	Пояснювальна записка	Наводиться опис проблематики, існуючих методів та підходів до вирішення завданння, пояснюється математичне підгрунтя методу пошуку, описується сам алгоритм пошуку, структура проекту та опис основного функціоналу
2	Керівництво користувача	Задається набір інструкцій та рекомендацій користувачу для роботи з програмою
Компоненти		
1	Функція fourierTransform	Математична підчастина алгоритму
2	Функція inverseFFT	Математична підчастина алгоритму
3	Функція fourierMultiplying	Загальна об’єднана математична алгоритму
4	Функція fourierComparison	Функція, включаюча математичний апарат та власне виконуюча порівняння рядків

Додаток 2.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

Текст програмного модулю

роботи на тему

“ШВИДКЕ ПЕРЕТВОРЕННЯ ФУР’Є В ЗАДАЧАХ ОБРОБКИ ТЕКСТІВ”

Виконав
студент групи ТВ-з61
кафедри АПЕПС
факультету ТЕФ
Джулай Володимир

Київ - 2020

Програмний код найважливіших функцій

```
void fourierTransform(vector<complex<double> >& complex_vector)
{
	unsigned int N = complex_vector.size();
	bit_reversal(complex_vector, N);

	unsigned k, bind_step, l, a;
	double complex_base;
	complex<double> w;
	complex<double> T;

	// ВИКОНАННЯ ШВИДКОГО ПЕРЕТВОРЕННЯ ФУР'Є З
	// ДОПОМОГОЮ МЕТОДІВ ДИНАМІЧНОГО ПРОГРАМУВАННЯ
	for(k = 2; k <= N; k <<= 1)
	{
		// ВИЗНАЧЕННЯ КОМПЛЕКСНОГО КОРЕНЯ ДЛЯ
		// МІНУС-  ОДИНИЦІ ЗГІДНО КІЛЬКОСТІ ЕЛЕМЕНТІВ ДЛЯ
		//ДАНОЇ ІТЕРАЦІЇ
		bind_step = k >> 1;
		complex_base = 3.14159265358979323846264338328L / bind_step;
		w = complex<double>(cos(complex_base), -sin(complex_base));
		T = 1.0L;
		for (l = 0; l < bind_step; l++)
		{
			// ЦИКЛ ПОПАРНОГО ЗАСТОСУВАННЯ ОПЕРАЦІЙ
			// "МЕТЕЛИКА" ЗІ ЗМІНОЮ КОМПЛЕКСНОГО
```

```
                    // КОЕФІЦІЄНТУ
                    for (a = l; a < N; a += k)
                    {
                        // ВИЗНАЧЕННЯ ПЕРВИННИХ ЗНАЧЕНЬ
                        // ОПЕРАНДІВ НА ОСНОВІ
                        // ПОПЕРЕДНЬОВНЕСЕНИХ ЗНАЧЕНЬ
                        complex<double> u = complex_vector[a];
                        complex<double> t = T * complex_vector[a + bind_step];
                        // ПЕРЕВИЗНАЧЕННЯ ОПЕРАЦІЯМИ “МЕТЕЛИКА”
                        // НОВИХ ЗНАЧЕНЬ ДЛЯ ПАРИ КОЕФІЦІЄНТІВ
                        complex_vector[a] = u + t;
                        complex_vector[a + bind_step] = u - t;
                    }
                    // ЗМІНА КОМПЛЕКСНОГО КОЕФІЦІЄНТУ
                    T *= w ;
                }
            }
        }

void inverseFFT(vector<complex<double> >&);

void fourierMultiplying(vector<int>& v_text, vector<int>& v_pattern,
vector<int>& v_result, bool self_compare)
{
        unsigned int N = v_text.size();
        vector<complex<double> > vcompl_result(N, 0);

        int i;
```

```
        // ПЕРЕТВОРЕННЯ ЦІЛОЧИСЕЛЬНОГО МАСИВУ У МАСИВ
        // КОМПЛЕКСНИХ ЧИСЕЛ
        vector<complex<double> >  vcompl_text(v_text.begin(), v_text.end());
        // ОТРИМАННЯ РЯДУ ФУР'Є ДЛЯ УМОВНО ТЕКСТОВОГО
        //  МАСИВУ
        fourierTransform(vcompl_text);

        // У ВИПАДКУ ОБЧИСЛЕННЯ ЗБІЖНОСТІ -
        //   ПОМНОЖИТИ КОЕФІЦІЄНТИ ПЕРЕТВОРЕННЯ ФУР'Є НА
        // СЕБЕ
        if(self_compare)
        {
                // multiplying polynomial with itself
                for(i = 0 ; i < N; ++i){
                        vcompl_result[i] = vcompl_text[i] * vcompl_text[i];
                }
        }
        // У ВИПАДКУ ПОРІВНЯННЯ З ТЕКСТОМ -
        // ОБЧИСЛИТИ ПЕРЕТВОРЕННЯ ФУР'Є ДЛЯ ЗРАЗКА, ПОТІМ
        // ПЕРЕМНОЖИТИ ПОІНДЕКСНО З РЕЗУЛЬТАТОМ
        // ПЕРЕТВОРЕННЯ ФУР'Є ДЛЯ ТЕКСТУ
        else
        {
                vector<complex<double> >  vcompl_pattern(v_pattern.begin(),
v_pattern.end()); //cout << "vcompl_pattern initialization\n";

                fourierTransform(vcompl_pattern);
```

```
            // multiplying two polynomials
            for(i = 0 ; i < N; ++i){
                  vcompl_result[i] = vcompl_text[i] * vcompl_pattern[i];
            }
      }

      //ЗАСТОСУВАТИ ЗВОРОТНЄ ПЕРЕТВОРЕННЯ ФУР'Є ДО
      // РЕЗУЛЬТАТУ ПЕРЕМНОЖЕННЯ ПЕРЕТВОРЕНЬ ФУР'Є ТЕКСТУ
      // ТА ЗРАЗКА ДЛЯ ОТРИМАННЯ РЕЗУЛЬТАТУ МНОЖЕННЯ
      // МНОГОЧЛЕНІВ
      inverseFFT(vcompl_result);
      //ВИДАЛИТИ ЗАЙВИЙ ЕЛЕМЕНТ
      vcompl_result.pop_back();
      // ОКРУГЛИТИ ТА ВИДІЛИТИ ЦІЛОЧИСЕЛЬНУ ОСНОВУ
КОМПЛЕКСНОГО РЯДУ МНОЖЕННЯ МНОГОЧЛЕНІВ
      for(i = 0; i < N; ++i)
            v_result[i] = round(vcompl_result[i].real());

}

// ОДИН З ВАРІАНІТВ ЗВОРОТНЬОГО ПЕРЕТВОРЕННЯ ФУР'Є
void inverseFFT(vector<complex<double> >& vc)
{
   // ЗМІНИТИ ЗНАК КОМПЛЕКСНОЇ ЧАСТИНИ НА ПРОТИЛЕЖНИЙ
   for(auto& x : vc) x = std::conj(x);

   // ЗАСТОСУВАТИ ПРЯМЕ ПЕРЕТВОРЕННЯ
```

```
    fourierTransform(vc);

    // ЩЕ РАЗ ЗМІНИТИ ЗНАК КОМПЛЕКСНОЇ ЧАСТИНИ НА
    // ПРОТИЛЕЖНИЙ
    for(auto& x : vc) x = std::conj(x);

    // ПОДІЛИТИ КОЖНЕ ЗНАЧЕННЯ НА КІЛЬКІСТЬ ЕЛЕМЕНТІВ
    // МАСИВУ
    for(auto& x : vc) x /= vc.size();
}

// ФУНКЦІЯ ПОРІВНЯННЯ ДВОХ ЧИСЛОВИХ МАСИВІВ НА ЗБІЖНІСТЬ
// З ЗАНЕСЕННЯМ У МАСИВ ІНДЕКСІВ ЗБІГІВ
int fourierComparison(vector<int>& v_text, vector<int>& v_pattern, vector<int>&
match_indices)
{
        // ЧАСОВА МІТКА
        auto start = chrono::steady_clock::now();

        // ЗМІННІ РОЗМІРІВ ТЕКСТУ ТА ЗРАЗКА
        int text_len = v_text.size();
        int pat_len = v_pattern.size();
        // ПЕРЕВІРКА НА ІСНУВАННЯ ТЕКСТУ ТА ЗРАЗКА
        if(text_len == 0 || pat_len == 0)
        {
                cout << "No input text or pattern for searching\n";
                return -1;
        }
```

```
        // ПЕРЕВІРКА НА ВІДПОВІДНІСТЬ ЗРАЗКА
        else if(pat_len > text_len)
        {
                cout << "Pattern line is longer than input text\n";
                return -2;
        }
        // ПРИВЕДЕННЯ ДО ОДНАКОВИХ РОЗМІРІВ ШЛЯХОМ
        // ДОДАВАННЯ НУЛІВ ТА ЗА ПОТРЕБИ ЗБІЛЬШЕННЯ РОЗМІРІВ
        // ДЛЯ ВІДПОВІДНОСТІ ЛОГАРИФМІЧНІЙ ФУНКЦІЇ ДЛЯ
        // ЗАСТОСУВСАННЯ МЕТОДУ ПОДІЛУ НА ДВА
        normalizeVectors(v_text, v_pattern);

        // ПЕРВИННЕ ПОРІВНЯННЯ ДЛЯ ПОДАЛЬШОГО
СПІВСТАВЛЕННЯ ОТРИМАНОГО РЕЗУЛЬТАТУ
        vector<int> pat_match_exposition(v_pattern.size(), 0);
        auto multiply_start = chrono::steady_clock::now();
        fourierMultiplying(v_pattern, v_pattern, pat_match_exposition, true);
        auto multiply_end = chrono::steady_clock::now();
        cout << "**********************************\n";
        cout << "fourierMultiplying vPattern function execution time: " <<
chrono::duration_cast<chrono::nanoseconds>(multiply_end -
multiply_start).count() << endl;


        // ПОРІВНЯННЯ З САМИМ ТЕКСТОМ
        vector<int> comparison_vec(v_text.size(), 0);
        multiply_start = chrono::steady_clock::now();
        fourierMultiplying(v_text, v_pattern, comparison_vec, false);
```

```
        multiply_end = chrono::steady_clock::now();
        cout << "**********************************\n";
        cout << "fourierMultiplying text&pattern function execution time: " <<
chrono::duration_cast<chrono::nanoseconds>(multiply_end -
multiply_start).count() << endl;


        // ПІДФУНКЦІЯ ПОРІВНЯННЯ КОЕФІЦІЄНТНОГО РЯДУ
        // МНОЖЕННЯ З ОПОРНИМ КОЕФІЦІЄНТОМ ЗБІГУ
        int pivot = pat_match_exposition[pat_len - 1];
        for(int pass_p = pat_len - 1; pass_p < text_len; ++pass_p)
        {
                if(pivot == comparison_vec[pass_p])
                        match_indices.push_back( pass_p - (pat_len - 1) );
        }
        // ДЕМОНСТРАЦІЯ РЕЗУЛЬТАТІВ
        if( match_indices.empty() )
        {
                cout << "No match\n";
                return 1;
        }
        else
        {
                cout<<"Match indices found at \n"; //**************************
                for(auto& x : match_indices) cout<<x<<" ";
//****************************************
                cout<<endl;
        }
```

```
        // КІНЦЕВА ЧАСОВА МІТКА
        auto end = chrono::steady_clock::now();
        cout << "**********************************\n";
        cout << "Fourier Search main function execution time in nanoseconds: " <<
chrono::duration_cast<chrono::nanoseconds>(end - start).count() << endl;

        return 0;
}
```

Додаток 3.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

Опис програмного модуля

роботи на тему

“ШВИДКЕ ПЕРЕТВОРЕННЯ ФУР’Є В ЗАДАЧАХ ОБРОБКИ ТЕКСТІВ”

Виконав
студент групи ТВ-з61
кафедри АПЕПС
факультету ТЕФ
Джулай Володимир

Київ — 2020

Анотація

Програма розроблена мовою Сі++, програма входить в один файл, не є розподіленою системою. Вона не сформована за принципами об’єктно-орієнтованого програмування, натомість використовує функціональний підхід. Драйверна функція реалізовує алгоритм пошуку на основі швидкого перетворення Фур’є як основне завдання даної роботи. Послідовність дій розбита на логічні блоки та втілена у якості підфункцій, що входять до складу основної функції. Програма призначена для простого використання з методами порівняння її швидкодії та можливостями для подальшого покращення.

Ключові слова: Швидке Перетворення Фур’є; Алгоритми пошуку в тексті за зразком.

Зміст

Загальні відомості

Програма називається fourier_search. Являє собою об’єктний файл для запуску на Linux-подібних операційних системах. Робота виконана з допомогою мови Ci++ без залучення додаткових фреймворків або баз даних. Програма працює з текстовими файлами типу .txt, вміст яких є набором символів, представлених в ASCII-таблиці.

Функціональне призначення

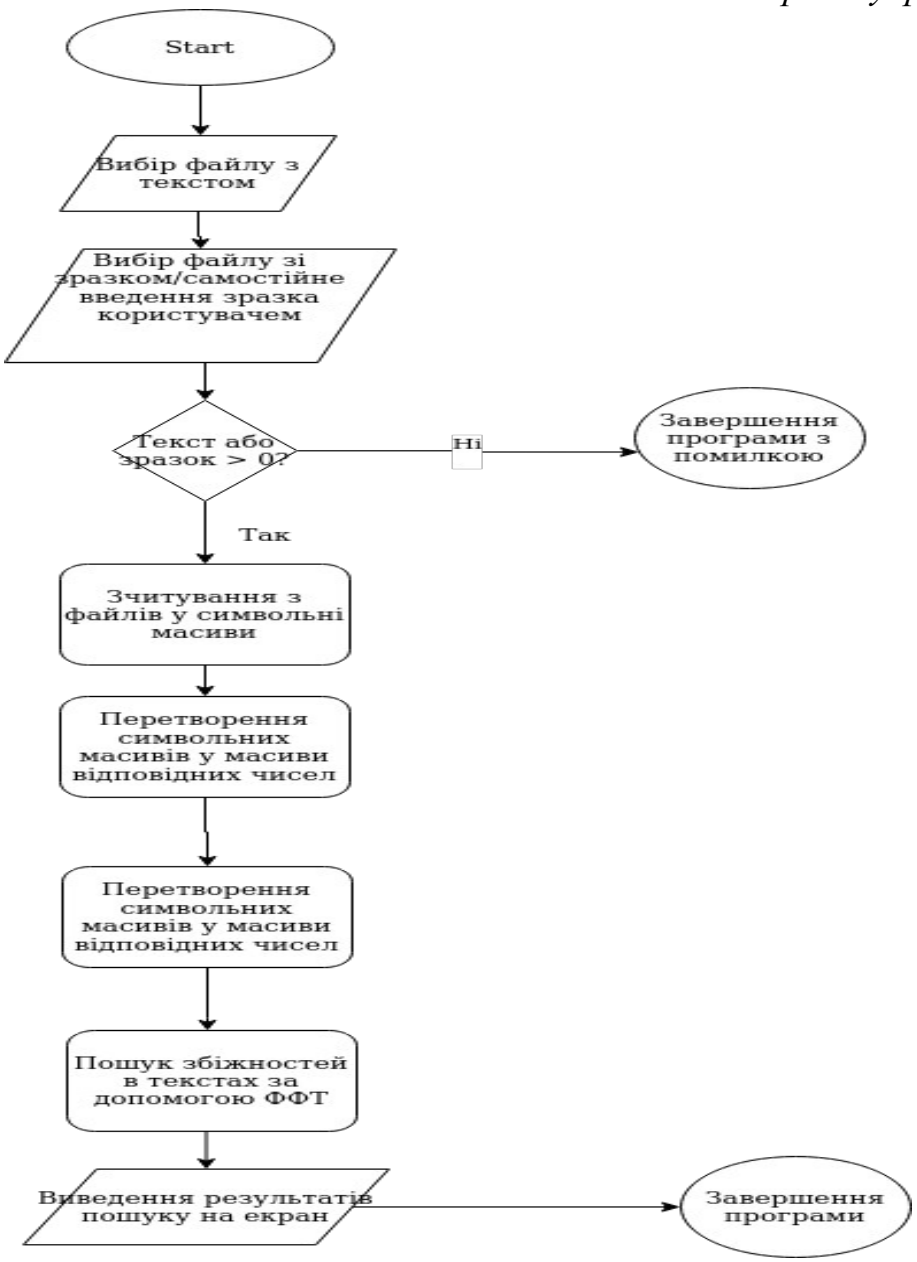
Програма розв’язує задачу пошуку текстового зразка в певному тексті. Її реалізація є предметом пошуків нових методів та алгоритмів для здійснення швидкого пошуку на великих масивах символьної інформації. Особливою потребою є реалізація алгоритму, який би справлявся з пошуком у тексті великого за обсягом зразка. Безумовно, існують простіші та ефективніші алгоритми для роботи з малими зразками, задаваними людиною. Однак їх ефективність падає при значних масивах даних, які надходять в певний програмний комплекс у якості вхідних даних з іншої програми або функції.

Дана робота призначається в першу чергу для розробників та тестувальників, які зможуть краще виявити сильні та слабкі сторони запропонованого алгоритму, та продовжать роботу в напрямку знаходження нового методу, що був би новаторським та конкурентноздатним водночас.

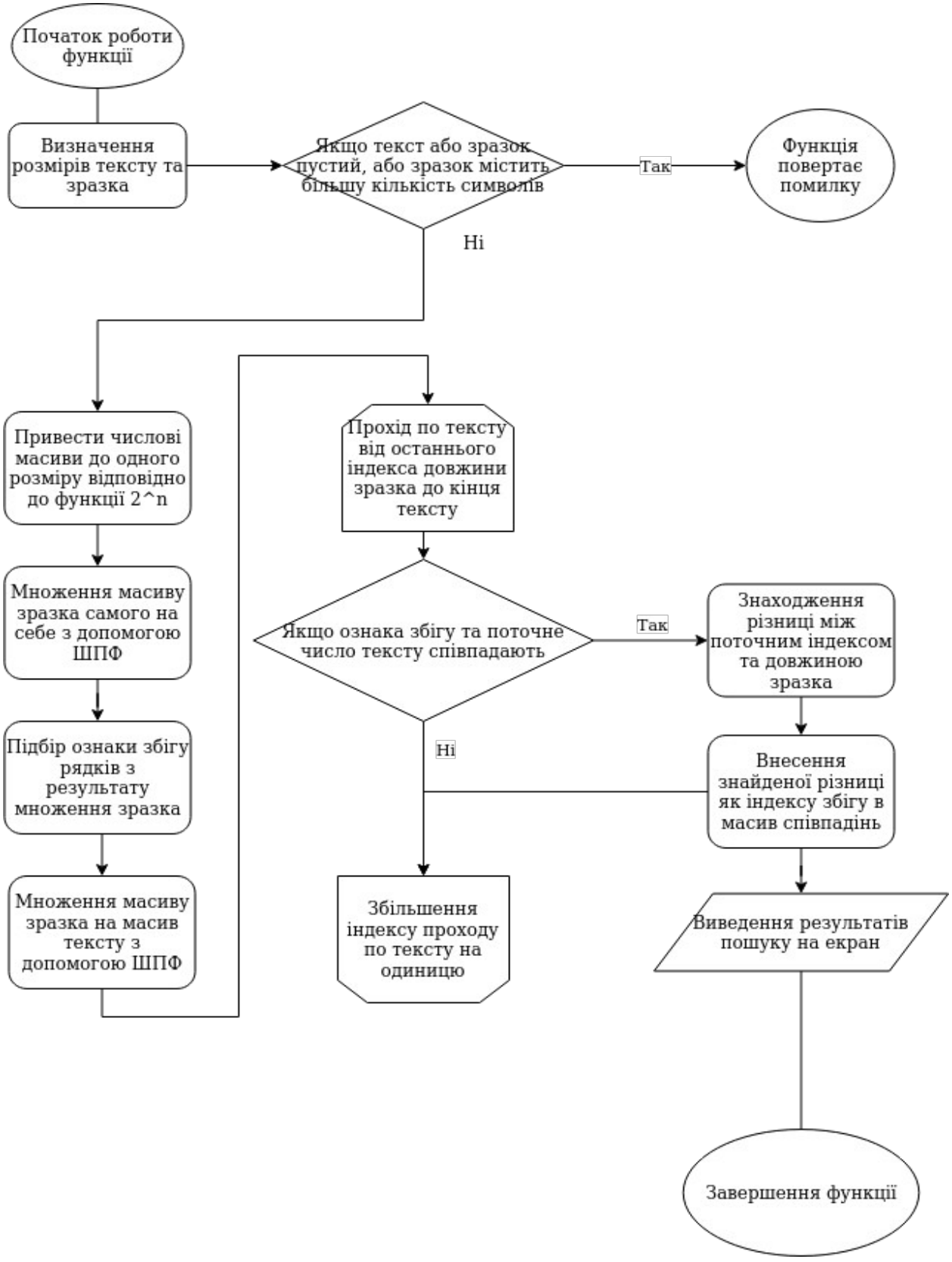
Опис логічної структури

В даній частині наведемо алгоритм роботи деяких важливих частин програми. Наведемо діаграму алгоритму роботи головної функції **main** даної програми, що готує дані для обробки пошуковою функцією fourierComparison, а також діаграму власне цієї пошукової функції, так як основний алгоритм пошуку збіжностей реалізований саме в ній.

*Блок-схема алгоритму функції **main***



Блок-схема функції fourierComparison



Використовувані технічні засоби, виклик і завантаження

При роботі програми в навчально-демонстративних цілях можуть використовуватися процесори з порівняно невеликим обсягом оперативної пам’яті, до 2 ГБ. При застосуванні для обчислень на великих обсягах даних може затребуватися більший об’єм пам’яті для проведення громіздких обчислень, звісно, якщо перевага в швидкодії буде доведена. Програма створювалася з допомогою засобів розробки на операційній системі Ubuntu, в кінцевому варіанті вона була вміщена в одномодульний файл, та скомпільована в об’єктний файл. Даний файл є виконуваним в середовищі Linux-подібних операційних систем (однією з яких є Ubuntu), відтак може легко пересилатися засобами електронного зв’язку як-то месенджери або електронна пошта, та запускатися з терміналу операційної системи локальної машини, на якому даний файл. Для використання файлу може знадобитися задавання у Властивостях файлу опції щодо використання даного файлу як виконуваного. Результуюча інформація після запуску та завершення роботи програми виводиться у термінал.

Вхідні та вихідні дані

Вхідні дані надходять в програму з власнophttps://vak.in.uaуч вибраних файлів користувача. Файли мусять обов’язково бути простого текстового формату з розширенням .txt, та містити символи виключно з таблиці символів ASCII. Таким чином, програма обробляє лише дані типу char, чий розмір займає 1 байт.

В результаті виконання програми всі отримані дані не зберігаються, але за потреби можуть бути сформовані у файл, або бути передані на вхід іншої програми. Ці дані є масивом значень індексів співпадінь, та можуть бути використані для візуального виділення або переходу на ці мітки в тексті в текстовому редакторі з графічною оболонкою. Програма завершується демонстрацією результатів пошуку збігів в тексті у вигляді масиву чисел, які є індексами точних збігів. У разі використання на місцях певних символів всередині зразка символів типу "*", під значення даного символу підійде одинарне значення будь-якого іншого символу, тому можливий випадок неточного охоплюючого пошуку з “неважливими” літерами. Окрім цього, програма демонструє результати виконання пошуку в тексті з допомогою іншого, класичного та перевіреного алгоритму. Важливою складовою результатів є обчислення сумарного часу виконання функцій, що використовують обидва алгоритми пошуку для порівняння їх результатів. Для отримання точних результатів необхідно провести тест з запуском того самого порівняння текстів певну кількість разів з подальшим обрахуванням середнього значення по результатах, так як час виконання на одній машині, не кажучи вже про різні машини, буде різний.